CROSSTALK

OPEN SOURCE SOFTWARE
Sharing From a Well of Ideas

1. REPORT DATE **JAN 2005**	2. REPORT TYPE	3. DATES COVERED **00-00-2005 to 00-00-2005**

4. TITLE AND SUBTITLE **Crosstalk: The Journal of Defense Software Engineering. Volume 18, Number 1, January 2005**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **OO-ALC/MASE,6022 Fir Ave,Hill AFB,UT,84056-5820**	8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**32**	

Departments

ON THE COVER
Cover Design by Kent Bingham.

Using Free Software Doesn't Mean It Won't Cost Anything

Each month Kent Bingham, our cover artist, provides us with at least three ideas as we plan a given issue's cover. One of Kent's ideas for this month was a selection of soups with the caption, "There really is a free lunch." As I thought about this option, I was worried how many of our readers would get this same idea regarding open source software. When deliberating using open source software, the user needs to understand that there are still costs that must be considered.

As with contemplating any new software product, the requirements for the software must be compared with its benefits and costs. Match your requirements with the software capabilities. If its features don't match all your requirements, you must decide if these are truly hard requirements or something you can live without. If they are hard requirements, are you prepared to develop the features yourself and add them to the software? What will it cost you to develop these required software upgrades? What will it cost to maintain your add-ons with each new upgraded software release? If the software is a tool that your organization will use to develop software, is the tool adequately documented so that the learning curve is reasonable? Are you willing to abide by the licensing agreement to share your improvements with the software sustainers?

We start this month's issue of CROSSTALK with an important policy released by the U.S. Air Force (USAF). As Michael R. Nicol explains in the abstract, the USAF believes we need to apply a renewed focus to software development and acquisition as we continue to work more with systems. The USAF plans to supplement the policy in future months with additional guidance, training, and other tools to support its successful implementation.

We begin our theme articles with an overview of open source software by David Tuma. In *Open Source Software: Opportunities and Challenges*, Tuma expands on the idea that while open source software has some great advantages, the user needs to be aware of challenges when deciding whether or not to use it.

In *Open Source Opens Opportunities for Army's Simulation System*, Douglas J. Parsons and Dr. Robert L. Wittman Jr. discuss their own twist to open source software. The OneSAF program has proven beneficial to the defense community and has even won the U.S. Government's Top 5 Quality Software Projects award. In this article, we learn that the software is being made available to the defense community and others who may have a valid need for it. However, given the military requirements, following all the defined requirements for open source software is not feasible.

Jonathan E. Shuster discusses another open source software product in *Introduction to the User Interface Markup Language*. This variation of the User Markup Language is available to the community and might meet your requirements.

We finish Hoyt Lougee's discussion on reuse in this issue with his follow-up article, *DO-178B Certified Software: A Formal Reuse Analysis Approach*. If you read Lougee's previous article last month, then you will know that the techniques discussed are applicable not only to DO-178B, but also are good overview ideas for any software reuse effort.

We conclude this issue with an Open Forum article by Michelle Levesque and Jason Montojo. In *Opening Up Open Source*, Levesque and Montojo bring to the forefront some of the difficulties encountered when trying to make full use of open source software.

As these articles show, open source software provides the opportunity for enhancing your own software without significant cost, but it is still not a free lunch. There are costs associated with deciding if the software meets your needs and enhancing it to implement missing features. It is also possible that a lack of documentation and training will hinder using the software effectively. As with any tool, open source software can make software development a less strenuous and expensive task if it is used with the proper expectations and oversight.

As we begin this new year, I would also like to thank CROSSTALK's Editorial Board (CEB), most of whom donate their time to help make CROSSTALK the best it can be. These reviewers strive to provide the authors with useful comments that will strengthen the articles and make them useful for our readers. A list of CEB members can be found on page 30.

Elizabeth Starrett

Elizabeth Starrett
Associate Publisher

Policy Note to Readers

Michael R. Nicol

Technical Advisor, Embedded Computer Systems Software Aeronautical Systems Center

U.S. Air Force

Section 804 of the fiscal year 2003 National Defense Authorization Act (Public Law 107-314) requires each military department to improve its software acquisition processes. The Air Force approach integrates Section 804 and ongoing systems engineering improvement activities to support our agile acquisition objectives of decreasing acquisition cycle time and improving our credibility in acquisition program execution. As a first step, we recently established policy to revitalize the software aspects of systems engineering. The policy identifies focus areas that we consider fundamental to developing realistic program baselines and promoting discipline in the acquisition and development of software-intensive systems. We plan to supplement the policy in the next few months with additional guidance, training, and other tools to support successful implementation of our acquisition improvement objectives.

UNDER SECRETARY OF THE AIR FORCE
WASHINGTON

SEP 2 0 2004

MEMORANDUM FOR SEE DISTRIBUTION   04A-003

SUBJECT: Revitalizing the Software Aspects of Systems Engineering
REFERENCE: Air Force Software-Intensive Systems Strategic Improvement Program (AFSSIP) memo dated 13 Jan. 2004.

In multiple programs across our acquisition communities, we have recognized systems engineering challenges over the past few years, and have taken steps to improve the implementation and effectiveness of our systems engineering processes.

This policy memorandum is intended to improve the efficiency and effectiveness of our acquisition processes and software management. These processes are applied as an integral part of our systems engineering and capability acquisition processes. To support our overall agile acquisition objectives, we expect you to address, as a minimum, the following software focus areas throughout the life cycle of your acquisition programs beginning with pre-Milestone/Key Decision Point A activities:

1. **High Confidence Estimates:** Estimate the software development and integration effort (staff hours), cost, and schedule at high (80-90%) confidence.

2. **Realistic Program Baselines:** Ensure cost, schedule, and performance baselines are realistic and compatible. Ensure the baselines support the disciplined application of mature systems/software engineering processes, and ensure software-related expectations are managed in accordance with the overall program's expectation management agreement. The program budget must support the high confidence estimates for effort (staff hours), cost, and schedule.

3. **Risk Management:** Continuously identify and manage risks specific to computer systems and software as an integral part of the program risk management process. Ensure the risks, impact, and mitigation plans are appropriately addressed during program and portfolio reviews.
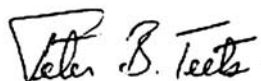
4. **Capable Developer:** Identify the software-related strengths, weaknesses, and risks; domain experience; process capability; development capacity; and past performance for all developer team members with significant software development responsibilities. Consider this information when establishing program baselines and awarding contracts, and throughout the program execution.

5. **Developer Processes:** Ensure the entire developer team establishes, effectively manages, and commits to consistent application of effective software development processes across the program.

6. **Program Office Processes:** Ensure the program office establishes and employs effective acquisition processes for software, is adequately staffed, and consistently supports the developer team in the disciplined application of established development processes.

7. **Earned Value Management Applied to Software:** Continuously collect and analyze earned value management data at the software level to provide objective measures of software cost and schedule. The Earned Value Management System should support and be consistent with the software effort and schedule metrics.

8. **Metrics:** Employ a core set of basic software metrics to manage the software development for all developer team members with significant software development/integration responsibilities. Guidance for the core metrics is provided in the enclosure. Programs are encouraged to implement additional metrics based on program needs.

9. **Life Cycle Support:** Address sustainment capability and capacity needs during the system design and development phase, and balance overall system acquisition and sustainment costs. Ensure you plan, develop, and maintain responsive life cycle software support capabilities and viable support options.

10. **Lessons Learned:** Support the transfer of lessons learned to future programs by providing feedback to center level Acquisition Center of Excellence (ACE) and other affected organizations. Lessons learned information includes original estimates and delivered actuals for software size, effort, and schedule; program risks and mitigation approaches; and objective descriptions of factors such as added functional requirements, schedule perturbations, or other program events that contributed to successes and challenges.

These focus areas will be incorporated as appropriate in your Systems Engineering Plan, Integrated Program Summary, or acquisition plans. We also expect you to address these focus areas as applicable during Acquisition Strategy Panels and PEO portfolio reviews. PEOs may tailor the implementation of these focus areas as required and the appropriate Acquisition Executive will be notified of all tailoring.

Sample language and additional guidance will be available in November 2004 in an Air Force Software Guidebook. Our POCs are Mr. Ernesto Gonzalez, SAF/AQRE, 703-588-7846, Ernesto.Gonzalez@pentagon.af.mil, and Maj Mark Davis, SAF/USAL, 703-588-7385, Mark.Davis2@pentagon.af.mil.

MARVIN R. SAMBUR
Assistant Secretary of the Air Force
(Acquisition)

PETER B. TEETS
Undersecretary of the Air Force

Open Source Software: Opportunities and Challenges

David Tuma
Software Process Dashboard Initiative

Much more than a buzzword, open source software is becoming an increasingly important part of the information technology environment. Many program managers, project managers, and developers in the Department of Defense and elsewhere are already familiar with open source; others may wonder how best to use open source in a project environment. In considering the opportunities presented by open source software, it is helpful to understand its origins as well as the challenges you may face in implementing it.

Whether you realize it or not, you rely on open source software every day. Open source software provides the underpinnings for the Internet, directs much of the world's e-mail traffic, and powers more than two-thirds of the world's Web sites [1].

The open source paradigm is based on the idea that software reuse need not stop at organizational boundaries. By sharing source code freely under a license that generously permits copying, modification, and redistribution, open source projects allow collaborative software development that benefits a larger community.

Although open source software has been in existence for decades, the arrival of the Internet has led to a veritable explosion in open source activity. The Internet has made it possible for developers around the world to discover each other, collaborate in real time, and share the works they create with others.

Organizations, businesses, and governments around the world are opening up to the possibilities provided by open source software. Although open source software has been in use for some time in the Department of Defense (DoD), a policy released in May of 2003 [2] officially put open source software on a level playing field with proprietary software.

If you have not explored open source software firsthand, you might be surprised by its diversity. Although media coverage commonly focuses on the Linux operating system and on server-based applications like Apache and MySQL, these applications are only the tip of the iceberg. System administration tools like Snort (an intrusion-detection tool), Nessus (a vulnerability scanner), and NetCat (a network debugging and mapping tool) help to manage computer networks. A wide variety of tools ranging from the Gnu's Not Unix (GNU) C Compiler to Perl (a popular programming language) to Eclipse (a Java integrated data environment developed by IBM) target software develop-

ment. An extensive array of reusable libraries and frameworks can save your software project time and money. Applications like Mozilla (a Web browser) and OpenOffice (an office productivity suite) address common end-user needs. And of course, the list goes on – one popular open source Web site alone lists more than 11,000 stable/mature open source projects [3].

Strengths of Open Source

Many organizations are initially attracted to open source products because they can generally be acquired for free. Momentarily deferring the debate about total cost of ownership until later in this article, removal of the initial procurement barrier can at times be a significant enabling factor for software use. For example, budget constraints have encouraged academia to adopt and support open source software for decades. With the increased public awareness of open source, public and private middle and secondary schools are beginning to investigate open source options, as well [4]. And small businesses may find open source software helpful in *leveling the playing field*.

Government organizations and contractors often discover different reasons to appreciate zero-cost licenses for open source software. For example, in 1999 the Census Bureau needed to create a Web site but had no official information technology budget to make it happen; staffers were able to build the Web site using open source applications and existing hardware, and the Web site is still in use today [5]. In addition, open source approaches can lower the monetary risk of experimenting with new technologies, effectively speeding the pace of technology adoption and supporting the collaborative development of new standards [1]. Moreover, the simplified licensing model of open source software can facilitate inter-agency sharing and reuse of developed solutions [6].

In fact, reuse is a central tenet of the

open source development paradigm. Open source software licenses (as defined by the Open Source Foundation [7]) explicitly target software reuse by permitting source code to be copied, modified, and distributed. When organizations and individuals share a common need, they can share and reuse entire open source products rather than independently developing redundant code. Common examples of this style of reuse include not only off-the-shelf open source products like the Apache Web server, but also application frameworks like Struts, and reusable libraries for performing tasks such as parsing eXtensible Markup Language or consuming Web services. Communities organized around this type of open-source reuse foster rapid innovation and progress, as many contributors can simultaneously develop improvements that benefit all users of the product.

The true strength of open source reusability, however, emerges when an organization or an individual has a unique need. An organization or individual with a new or unmet need is free to modify an open source product[1] to meet that need, potentially reusing thousands of lines of code. The benefits of reuse in such a scenario are well understood, saving countless hours of development, testing, and maintenance. Contributing such an enhancement back to the open source community can benefit other organizations with similar needs [8].

Open source software promotes reuse in another, unexpected way through code transparency. Inadequate documentation has long been identified as a significant barrier to software reuse; in addition, subtle misunderstandings between developers on either side of a code boundary can lead to insidious errors. In the absence of flawless code and impeccable documentation, the freedom to examine source code can mean the difference between a useful, reusable library and a baffling *black box*.

When code in an open source library

behaves unexpectedly, a developer can peer into it to understand its operation and true intent, and determine whether the defect lies in the library or in their mental model of it. If the defect is in the library, the developer can either notify the original author or fix it himself or herself. Otherwise, the developer can gain a better understanding of the library and correct his or her code to use it properly. In either case, transparency across the code boundary helps to improve the quality of the overall system.

In the broader pursuit of software quality, many open source projects succeed by leveraging code review practices on a massive scale. Studies have demonstrated that code reviews (analyzing source code for problems) can remove defects much more effectively than testing (running an application and watching for incorrect behavior). In a closed source environment, only internal developers can perform code reviews, while the larger user community is constrained to *black-box testing*. Open source projects remove this limitation, freeing any end user to participate in the code review process.

While there is a practical limit to the number of software developers that can work together on a single team (because communication needs increase exponentially with the number of developers), there is almost no limit to the number of people who can simultaneously review code and test an application [9]. Accordingly, successful open source projects like Linux harness the skills of their large user base to perform massively parallel code reviews. Even brute-force testing methods can prove effective in improving product quality when thousands of individuals participate, each testing a product from his or her own unique perspective. When the conditions are right, open source projects can successfully employ these techniques to develop very high-quality products [10].

Security of Open Source Software

Open source proponents cite the collaborative review process as a major strength of the open source paradigm, ideally suited for producing highly reliable, secure code. Nevertheless, the security of open source software (and its comparison to the security of proprietary software) is a hotly debated topic.

For example, open source critics have recently questioned the use of Linux for defense systems. In a recent press release, Green Hills Software, Inc.'s Chief Executive Officer Dan O'Dowd stated, "The open source process violates every principle of security. It welcomes everyone to contribute to Linux. Now that foreign intelligence agencies and terrorists know that Linux is going to control our most advanced defense systems, they can use fake identities to contribute subversive software that will soon be incorporated into our most advanced defense systems" [11].

Other experts quickly responded to O'Dowd's claims, labeling them *fear, uncertainty, and doubt*. Citing Easter eggs, backdoors, spyware, and malware, they pointed out that proprietary software could just as easily contain illegitimate code. Citing the rigorous public review process used to approve Linux code, they argued that a *foreign intelligence agency* could more easily infiltrate a commercial development project

---

> *"An organization or individual with a new or unmet need is free to modify an open source product to meet that need, potentially reusing thousands of lines of code."*

---

than slip malevolent code under the noses of hundreds or thousands of watching reviewers. If one is truly worried about such malicious code, they argued, open source development is a better approach than closed source development since it permits anyone to perform his or her own independent review [12, 13].

Experts on both sides of the open source security debate contribute many compelling arguments. The bottom line, however, is that open source software is not automatically more or less secure than proprietary software [14]. Both development approaches have their strengths and weaknesses, but neither automatically produces more secure code than the other. Unfortunately, impassioned people on both sides of the debate regularly make broad, unconditional assertions about the relative security of open source and proprietary software. Although such statements certainly keep the debate interesting (and make for colorful news items), objective analyses are more useful. An astute

observer should be skeptical of sweeping generalities and dig deeper to find impartial expert analysis.

Using Open Source Software

Many people may have preconceived ideas about potential uses of open source software. With the variety of products available, however, there are many ways projects might consider using open source (including but not limited to the following):

- Deploy onto *off-the-shelf* open source server software (such as Linux, Apache, or MySQL).
- Reuse open source architectural frameworks (such as Struts, Spring, or Zope).
- Make use of open source development tools (such as Ant, Eclipse, or CVS [Concurrent Versions System]).
- Leverage reusable libraries (such as Xalan, OpenSSL, or GTK+).

Like the DoD, many organizations currently have policies that permit using open source software when it meets applicable requirements and provides the best value for the money. Thus, project managers considering using open source software must be prepared to analyze all the options available – both proprietary and open source – and determine which product provides the best value within the requirements for the project at hand. Project managers can immediately encounter several challenges relating to open source products.

For example, project managers may have difficulty discovering what open source products are available. Unlike proprietary alternatives, open source products rarely have budgets for advertising and marketing. And while mainstream media often includes news items on *flagship* open source products like Linux and Apache, you are unlikely to find information on frameworks, libraries, tools, or less common applications.

Fortunately, many Internet resources are available. Two of the largest Web sites are <freshmeat.net> [15] and <source forge.net> [3]; both list tens of thousands of open source software items in a categorized and searchable format. Keep in mind that open source (like technology in general) can progress at a remarkable rate, and new open source products and frameworks can seemingly appear overnight. Similarly, an open source project that might have had too many rough edges six months ago may now exceed your needs today. To keep abreast of these changes, software developers may find <slashdot.org> [16] to be a useful source

of product announcements.

With a list of potential open source and proprietary options ready, the daunting challenge of determining best value begins (see Table 1). This project management task has never been simple, but open source introduces many additional challenges.

For example, when considering only proprietary options, managers might glean information from the *market price* of a product. It might be a safe working assumption that a $50,000 product has more features than a $500 product. Where, then, does an open source product fit in the list? In a similar vein, managers can often look at *market share* statistics for proprietary products to see which ones are most popular. Unfortunately, this is often impossible for all but a handful of open source products. Although the Hyper Text Transfer Protocol makes it possible to estimate the number of Apache Web servers in use around the world, there is almost no way to determine how many people are using Linux, OpenOffice, or many of the tens of thousands of other open source applications in existence. These challenges make it more difficult to build a *short list* of products to choose from.

As a result, finding the *best value* comes down to a lot of research, legwork, and analysis. Find people within your organization and elsewhere who are using the products, and draw upon their experiences for pragmatic advice. Look for reviews in online publications. And above all, be wary of sweeping claims that *open source software is better/worse than proprietary software in category XYZ*. Although it is tempting to listen to such claims (because they would certainly simplify your decision-making

Table 1: *Checklist for Comparing Software Options*

Checklist for Comparing Software Options
Cost-Related Factors
☐ Software Costs (purchase, upgrades, licensing)
☐ Hardware Costs (purchase, upgrades)
☐ Staffing Costs (internal and contract staff)
☐ Internal and External Support Costs (installation, maintenance, troubleshooting)
☐ Indirect Costs (downtime, training)
Qualitative Factors
☐ Customizability/Flexibility
☐ Availability/Reliability
☐ Interoperability
☐ Scalability
☐ Performance
☐ Security
☐ Manageability/Supportability
☐ Expected Lifetime
Source: "A Business Case Study of Open Source Software" [17]

process), they rarely withstand careful scrutiny. In truth, comparisons must be made on a case-by-case basis, taking into consideration not only the products in question but also the unique requirements of your project. Here are some items to include on your comparison checklist (see [17] for a more thorough checklist):

- **Requirements.** This is, of course, one of the most important characteristics to consider: Does the software provide the functionality your project needs? If an open source product is missing a small function you need, is it possible and cost effective to add the feature yourself (keeping life-cycle costs in mind)? Does it meet your project's requirements for performance, quality, reliability, scalability, and security?
- **Licensing Restrictions.** Open source software is distributed under various licenses with differing terms. If some of these are incompatible with your project's target environment (see discussion below), they should be eliminated early in the selection process.
- **Support.** What quality of support is available for the various options on your list, and how much does that support cost? Support for open source products may be available from the original developers or from third parties. If third-party support is not available, you can gauge the level of support from the open source community by scanning related help forums, bug trackers, and mailing list archives. Throughout the past months, have user cries for help gone unanswered, or have they been addressed quickly? Larger open source projects – especially the *flagship* open source products like Linux and Apache – often have excellent support [18, 19, 20]. Support for smaller projects may be lacking – especially if the project is no longer under active development. In such cases, you will need to estimate how much it would cost to support the application yourself.
- **Documentation.** Is the product adequately documented? Does the documentation appear up-to-date? Are third-party books on the product available?
- **Maintenance Costs.** Will the deployed product be easy to maintain? For example, will it require monitoring and patching, and are tools available to help with those tasks?
- **Skills.** Do members of your team have expertise with the products in question? If not, is training available (either online or from a third party)? If you plan to outsource or subcontract project work or maintenance, are experienced contractors/integrators available?
- **Warranties.** Open source software typically comes with no warranties, although third-party warranties may be available. How do these compare with the warranties for the proprietary software choices on your list?
- **Vendor Lock-In.** Is the product standards-based, or does it lock you into a particular proprietary solution? Although most open source products are vendor-neutral, not all are. If technology neutrality is important to your project, examine your options carefully.

Ultimately, many of these considerations roll up into the larger concept of *total cost of ownership* (TCO). TCO has received a lot of media attention lately, and will continue to be a source of debate; like all of the characteristics above, however, TCO must be evaluated on a case-by-case basis. Some projects will see a lower TCO from proprietary solutions, while some will see a lower TCO from open source products. And with certainty, the types of projects that benefit from each will change over time, as both proprietary and open source products move forward.

If your research and analysis lead you to select an open source product for your project, it is, of course, imperative that you understand and respect the license terms of that open source software[2]. Because open source software is generally available at no cost, people often mistakenly assume that the code is in the public domain and can be used without restrictions. On the contrary, open source software is generally distributed under one of the licenses approved by the Open Source Initiative [7].

By definition, open source licenses universally grant broad permission to copy, modify, and distribute source code and compiled binaries, as long as the terms of the license agreement are respected. In many cases, these terms are very simple to comply with; for example, they may require a specific copyright notice and disclaimer to be included in the end-user documentation of a product that redistributes an open source library. Of course, the terms vary from license to license, and dozens of open source licenses are in active use today, so it is important to carefully read, understand, and comply with the licenses of any open source products you use. Fortunately, this task is not as difficult as it sounds, since a small number of licenses (listed in Table 2) cover as much as 90 percent of the open source software currently available.

If you modify an open source product

or compile it into a larger program, additional licensing terms may apply. For example, some licenses will require your modifications to be released back to the open source community. Ensure that you read the license carefully and understand its requirements. In particular, most organizations will want to avoid compiling and linking code distributed under Gnu's Not Unix (GNU) General Public License into a larger project [21].

When using, modifying, or enhancing open source software, it is also important to understand any applicable restrictions that stem from organizational policy, contractual requirements, and the like. For example, if your organization forbids any external release of code, and a particular open source product requires distributed code modifications to be released as open source, then you may not be able to modify that library and still meet all your legal obligations[2]. From a project management standpoint, it is best to be aware of such constraints before heading down a dead-end path.

Participating in the Open Source Community

As awareness of open source software grows, and as open source usage becomes a more common part of everyday software development, more and more individuals and organizations wonder how they can get involved with the open source community.

Some organizations have successfully embraced the open source development model for their managed projects, accepting code contributions from external developers. However, since external developers may not be accountable to the internal project goals, this approach introduces risks that most projects are not able to accept. Fortunately, there are still many other creative ways to work with the open source community.

Perhaps the simplest way to participate in the open source community is to provide feedback and bug fixes to open source projects. If your project uses an open source product (whether it be an operating system, an application, a framework, a reusable library, or some other product), take the time to thank the developers who created it. In many cases, this thanks is the only payment they receive for their efforts. If you discover and/or fix a bug in the product, you can benefit the entire community by sharing your discovery or patch with the product developers.

Another way to participate in the open source community is to contribute

enhancements to an open source product. If you discover that a particular open source product meets most (but not all) of your needs, and decide (through due diligence) that the best course of action for your project is to extend and enhance the open source product, consider contributing these enhancements back to the community when your project is done. Many commercial and government projects participate in open source in this way.

Some organizations have gone even further, contributing completed projects in their entirety to the open source community. In addition to the obvious benefits of reuse, organizations have discovered other unexpected benefits as well – for example, reduced life-cycle costs as open source developers begin fixing bugs and adding features [22]. Both government and corporate supporters of open source are increasingly using this approach.

Summary

Open source will continue to be an important part of the software landscape for years to come. Although misconceptions and misinformation often confuse the decision-making process, careful analysis can indicate where using open source is appropriate. Understanding the issues and opportunities inherent in open source is the first step in using it effectively to deliver maximum value for your project, your organization, and your clients.◆

References

1. Fordahl, Matthew. "Open-Source Software a Big Tech Player." AP Online 16 July 2004 <www.bizreport.com/news/7684>.
2. Stenbit, John P. "Open Source Software (OSS) in the Department of Defense (DoD)." Washington, D.C.: Defense Information Systems Agency, 28 May 2003 <http://iase.disa.mil/oss-in-dodmemo.pdf>.
3. Open Source Technology Group. SourceForge.net 12 Sept. 2004 <http://sourceforge.net>.
4. Surran, Michael. "Making the Switch to Open Source Software." T.H.E. Journal 31.2 (Sept. 2003): 36+ <www.thejournal.com/magazine/vault/a4499.cfm>.
5. Zieger, Anne. "Open-Minded: Government Agencies Are Overcoming Obstacles to Open Source." Government Enterprise 2 June 2002 <www.governmententerprise.com/showArticle.jhtml?articleID=17501499>.
6. Gallagher, Peter. Public InfoStructure – Inevitable Evolution: Unlocking Innovation for the Business of Government. Proc. of the Third Annual Open Source in Government Conference, George Washington University, Washington, D.C., 16 Mar. 2004 <www.egovos.org/Conferences/March_2004_Presentations>.
7. Perens, Bruce. The Open Source Initiative. Vers. 1.9. Open Source, 19 Oct. 2001 <www.opensource.org>.
8. Pavlicek, Russell. "Open Source Perspective: Open Source Origins." Processor 25.34 (22 Aug. 2003): 6 <www.processor.com/Editorial/article.asp?article=articles/p2534/06p34/06p34.asp>.
9. Raymond, Eric S. "The Cathedral and the Bazaar." Free-Soft.Org, 22 Nov. 1998 <www.free-soft.org/literature/papers/esr/cathedral-bazaar/cathedral-bazaar.html>.
10. United Nations. E-Commerce and Development Report 2003. United Nations Conference on Trade and Development, New York and Geneva, 2003: Chap: 4 "Free and Open Source Software: Implications for ICT Policy and Development." <www.unctad.org/en/docs/ecdr2003ch4_en.pdf>.
11. Green Hills Software. "Using Linux Software in Defense Systems Violates Every Principle of Security Says Green Hills Software's CEO and Founder." Santa Barbara, CA: Green Hills Software, 8 Apr. 2004 <www.ghs.com/news/20040408_AFEI.html>.
12. Groklaw. "CEO's of LynuxWorks and FSMLabs Reply to Green Hills' FUD." Groklaw. Ed. Pamela Jones. 11 Apr. 2004 <www.groklaw.net/article.php?story=20040411073918151>.
13. Singh, Inder. "LynuxWorks CEO, Dr. Inder Singh, Challenges Misrepresentative Claims Regarding Security in the Military." San Jose, CA: LynuxWorks, 2004 <www.lynuxworks.com/corporate/press/2004/linux-secure-military.php>.
14. Wheeler, David A. Open Source Software (OSS) and Security. Proc. of the Third Annual Open Source in Government Conference. George Washington University, Washington, D.C., 15-17 Mar. 2004 <www.egovos.org/Conferences/March_2004_

Commonly Used Open Source Licenses
- GNU General Public License
- GNU Lesser General Public License
- Berkeley Software Distribution License
- Artistic License
- Apache Software License
- Massachusetts Institute of Technology License
- Mozilla Public License

Table 2: *Commonly Used Open Source Licenses*

Presentations>.

15. Open Source Technology Group. Freshmeat.net 12 Sept. 2004 <http://freshmeat.net/about>.
16. Open Source Technology Group. Slashdot.org. Eds. Rob Malda, Jeff Bates, et al. 12 Sept. 2004 <http://slashdot.org/about.shtml>.
17. Kenwood, Carolyn A. "A Business Case Study of Open Source Software." Bedford, MA: The MITRE Corporation, July 2001 <www.mitre.org/work/tech_papers/tech_papers_01/kenwood_software/index.html>.
18. King, Julia. "A Sunny Forecast For Open Source." Computerworld 26 Apr. 2004 <www.computerworld.com/industrytopics/travel/story/0,10801,92583,00.html>.
19. Rapoza, Jim. "Can Open Source Provide Adequate Support?" eWeek 19 Apr. 2004 <www.eweek.com/article2/0,1759,1569380,00.asp>.
20. Dickerson, Chad. "CTO Connection: Open Source for a Song." InfoWorld 15 Aug. 2003 <www.info world.com/article/03/08/15/32OPconnection_1.html>.
21. Wacha, Jason B. "Open Source, Free Software, and the General Public License." Computer and Internet Law 20.3 (1 Mar. 2003): 20+.
22. Boos, Paul M. Using and Contributing to the Open Source Community While Supporting the Government. Proc. of the Third Annual Open Source in Government Conference. George Washington University, Washington, D.C., 15-17 Mar. 2004 <www.egovos.org/Conferences/March_2004_Presentations>.

Notes

1. In fact, the open source movement traces its origins (through the *free software* movement) to Richard Stallman's desire to enhance a proprietary printer driver [8].
2. The author of this article is not a lawyer. The information provided in this article is for informational purposes only and should not be construed as legal advice.

About the Author

David Tuma is the lead developer for the Software Process Dashboard Initiative, creating open source tools to support high-maturity software development processes. He first encountered open source software as a student at the Massachusetts Institute of Technology, and again later as a captain in the United States Air Force. As a strong supporter of open source, Tuma has been developing open source software on his own time for the past 10 years.

Software Process Dashboard Initiative
1645 E. HWY 193, STE 102
Layton, UT 84040-8525
Phone: (801) 771-4100
Fax: (801) 728-0595
E-mail: tuma@users.sourceforge.net

WEB SITES

Open Source
www.opensource.org
The Open Source Initiative (OSI) is a non-profit corporation dedicated to managing and promoting the open source definition for the good of the community, specifically through the OSI Certified Open Source Software certification mark and program. You can read about successful software products and about OSI's certification mark and program on the Web site.

SourceForge.net
http://sourceforge.net
SourceForge.net is an open source software development Web site maintaining one of the largest repositories of open source code and applications available on the Internet. SourceForge.net provides free services to open source developers.

GNU Operating System – Free Software Foundation
www.gnu.org
The GNU [GNU's Not Unix] Project was launched in 1984 to develop a complete free software, Unix style operating system: GNU (pronounced guh-noo). The Free Software Foundation is the principal organizational sponsor of the GNU project.

National Technology Alliance
www.nta.org
The National Technology Alliance (NTA) is a U.S. government program established in 1987 to influence commercial and dual-use technology development with an emphasis on meeting national security and defense technology needs. The NTA's goal is to partner commercial technology solutions to government user technology needs and then create new or enhanced commercial products where the cost of development is leveraged across a broad user community.

Open Source Software Institute
http://oss-institute.org
The Open Source Software Institute is a non-profit organization comprised of corporate, government, and academic representatives whose mission is to promote the development and implementation of open source software solutions within U.S. federal and state government agencies and academic entities.

Samba
http://us4.samba.org
Samba is an open source/free software suite that provides seamless file and print services allowing for interoperability between Linux/Unix servers and Windows-based servers. Samba is freely available under the GNU General Public License. Samba is software that can be run on a platform other than Microsoft Windows such as Unix, Linux, IBM System 390, OpenVMS, etc.

freshmeat
http://freshmeat.net
freshmeat maintains one of the Web's largest index of Unix and cross-platform software, themes and related eye-candy, and Palm OS software. Thousands of applications and links to new applications are added daily. Each entry provides a description of the software, links to download it and obtain more information, and a history of the project's releases so readers can keep up-to-date on the latest developments.

Open Source Opens Opportunities for Army's Simulation System

Douglas J. Parsons
Program Executive Office

Dr. Robert L. Wittman Jr.
MITRE Corporation

The One Semi-Automated Forces (OneSAF) Objective System is the U.S. Army's next-generation, entity-level, simulation system planned to provide a comprehensive set of tools supporting computer-based simulation event setup, execution, and review. Postured as an open-architecture, open-source application, the OneSAF program will put this software into the hands of a vast number of developers throughout the Department of Defense with the intent of creating unprecedented participation across the modeling and simulation community to include multi-service, international, industry, and academia experts in the evolution of the OneSAF system. This article describes the factors that led OneSAF to an open source development methodology, the open source principles OneSAF is supporting, and the key processes and tools supporting the open source development.

Since its beginnings in 1991 with a small group of programmers – some considered fanatics – Linux has become a force of hundreds of thousands [1]. The value of open source computing is found in the ability to leverage the talents and resources of an entire community. The Program Executive Office for Simulation, Training, and Instrumentation (PEO STRI) is positioning its next-generation constructive simulation to provide this same benefit to the Army's modeling and simulation (M&S) community.

The One Semi-Automated Forces (OneSAF) Objective System (OOS) is being developed to primarily serve training audiences, research scientists, and acquisition analysts. The OOS will also provide embedded simulation capabilities as part of the Army's Future Combat Systems. Once fielded in fiscal year 2006, the OOS will not only be used by the Army, but also will serve multi-service, international, industry, and academic organizations. Releasing source code to such a vast network of developers will certainly reap benefits for the Department of Defense (DoD) M&S community as a whole; however, distributing source code alone will not provide the optimal mechanism for a community to work together.

Initial efforts focused on developing a capable, robust, and extensible architecture supporting a toolkit that will allow users to grow the baseline. Active program office support, tools, and processes are also necessary to foster communication and increase the likelihood that community-developed capabilities will be integrated and shared with other users and developers. Finally, growing OOS product line capabilities will not be limited only to skillful Java programmers: A software toolset will allow a user who is not a programmer to build military entities, units, and their respective activities.

OneSAF Background

The OOS is the U.S. Army's next-generation simulation system that can represent a full range of military operations, systems, and control processes. It will accurately and effectively represent specific activities of combat; command, control, communications, computers, and intelligence; combat support; and combat service support. It is an entity-level simulation, meaning that it can simulate the activities of individual combatants or vehicles (as opposed to aggregate-level simulations, which represent combatants and vehicles as groupings). It will also provide the appropriate representations of the physical environment (e.g., terrain features, weather, illumination, etc.) and its effect on simulated activities and behaviors.

One aspect that makes the OOS unique among Army simulations is its design for use by three distinct Army M&S domains. Specifically, the Advanced Concepts and Requirements (ACR) domain uses M&S for experimentation and analyses on Army doctrine and force-related concepts. The Research, Development, and Acquisition (RDA) domain uses M&S for acquisition analyses focused on equipping and supporting currently fielded and future forces. Finally, the Training, Exercises, and Military Operations (TEMO) domain employs M&S to train the force. It does so using live simulation (actual equipment on training ranges), virtual simulation (immersing the trainee into a synthetic environment), and constructive simulation (war games using computer-generated forces).

It Is About Saving Resources

The OneSAF program concept originated in January 1996 following an extensive study concluding that the Army was caught in a wasteful spending cycle, making identical or similar enhancements to many legacy simulations across three different user domains. In May 1997, the deputy commanding general for Training and Doctrine Command approved the Mission Needs Statement for OneSAF, which stated:

> The need for OneSAF capabilities is not a response to a specific warfighting threat against the force; the need is driven by the guidance to reduce duplication of M&S investments, foster interoperability and reuse across M&S domains, and meet the M&S requirements of the future force. [2]

The Army decided the best approach for overcoming the problems associated with the multitude of aging simulations was to create a single, general-purpose, entity-level simulation and associated simulation event support tool [3].

Lessons Learned From a Legacy Simulation

The OneSAF program has drawn many lessons from the now-retired Modular Semi-Automated Forces (ModSAF) program. While the ModSAF simulation was nowhere near providing OOS' required capabilities, it was an entity-level military simulation serving a multi-domain M&S community. If not for the decision to release the source code, ModSAF would likely have been relatively unknown.

Funded by the Defense Advanced Research Projects Agency in the early 1990s, ModSAF was developed to facilitate synthetic environment research in support of distributed interactive simulation applications. Word of the availability of source code quickly spread through the M&S community, and requests for the software steadily grew up to the point of its retirement in 2002. By its end, more than 200 organizations had placed requests for the source code. The OneSAF Program Office reaped many lessons learned from the ModSAF program – those worth continuing as well as those that needed improvement.

Two ModSAF characteristics deemed critical to the success of OneSAF include the release of source code and the responsibility to provide services to facilitate and

enhance communications among the OneSAF user community. How OOS embraces these characteristics is described in the following paragraphs.

Open Source and OneSAF

Releasing source code as part of DoD applications raises numerous questions ranging from security concerns to baseline configuration management to cooperative development and, finally, integration. To address these concerns and to abide by DoD acquisition guidelines, OneSAF necessarily qualifies its definition of open source development.

For the vast majority of organizations that will request the OneSAF baseline, the distribution process will be much like that employed with ModSAF where the program manager (PM) of OneSAF distributes the baseline with source code at no cost. This is a condition where OneSAF aligns directly with a primary tenet of open source software as defined by the Open Source Initiative; however, there are key distinctions between the open source tenets and the OneSAF distribution model. The Open Source Initiative defines open source as software that provides the following rights and obligations [4]:

a) No royalty or other fee imposed upon redistribution.
b) Availability of the source code.
c) Right to create modifications and derivative works.
d) May require modified versions to be distributed as the original version plus patches.
e) No discrimination against persons or groups.
f) No discrimination against fields of endeavor.
g) All rights granted must flow through to/with redistributed versions.
h) The license applies to the program as a whole and to each of its components.
i) The license must not restrict other software, thus permitting the distribution of open source and closed source software together.

Of these, a, b, c, g, and h apply to the OneSAF distribution process. While not classified, the OOS will have content (e.g., data, algorithms) deemed sensitive by the U.S. Department of the Army. The baseline cannot be freely distributed as defined for open source due to security reasons. As such, PM OneSAF must be selective in the distribution of the OOS baseline. Essentially, there are two basic commitments made when a user signs a OneSAF distribution agreement:

1. Authorization to redistribute the baseline is restricted to PM OneSAF.
2. Users who develop new functionality into the OneSAF baseline agree to provide those capabilities back to PM OneSAF for possible reintegration.

These constraints offer advantages across the OneSAF user community. Facilitating distribution through a single focal point allows the PM to have knowledge of whom and how users intend to use the baseline. This knowledge will enhance the ability to identify and integrate useful community-developed capabilities into future baseline releases.

Helping to Communicate

In light of the restrictions OneSAF imposes on pure open source distribution, the OneSAF leadership felt compelled to provide communication-enhancing tools and processes that were seen as critical to the

Table 1: *Enabling Tools for OneSAF Open Source Development*

Tool	Description
Web-Based OneSAF Objective System Development Portal	The cornerstone of the OneSAF development environment is <https://www.OneSAF.net>. It is a secure Web site that houses technical information and historical and current programmatic, organizational, and task order structure. Technical information from architectural designs down to the Application Programmer's Interface (API) descriptions can be found on the site. The API descriptions are provided by automated code scrappers that generate Javadocs on a periodic basis. User ID and password are required.
Apache HTTPS Server	The Apache server <www.apache.org> provides a Web server for the OneSAF.net environment.
Mailman	Distributed asynchronous discussions and archiving is provided via e-mail using the Gnu's Not Unix free, open source product Mailman <www.gnu.org/software/mailman>. Mailman provides an integrated Web environment for managing e-mail discussions and e-newsletter lists. It offers a complement of mail list functionality, including built-in archiving, automatic bounce processing, content filtering, digest delivery, spam filters, and Web-based list administration.
Concurrent Versioning System	Configuration management and revision control processes and services are built around the Concurrent Versioning System (CVS) <www.cvshome.org>. CVS version 1.10.8 is freely available open source software, and provides revision control for all software development and Web-based information developed and used by the OneSAF team.
The Dynamic Object-Oriented Requirements System	Automated support for requirements management and tracking is provided by the Dynamic Object-Oriented Requirements System (DOORS) <www.telelogic.com>. Although neither freely available nor open source, this automated tool supports the requirements-driven OneSAF development process. DOORS version 7.0 provides automated support for OneSAF's rigorous requirements analysis and tracking process and is accessible to all task order participants within the OneSAF Integration and Development Environment building. DOORS allows requirements storage and retrieval, and maintains linkages between user, system, and software requirements.
Together Control Center	Automated software design and development support is provided by the Together Control Center (TCC) version 6.0 <www.togethersoft.com>. The TCC is neither free nor open source, but is necessary to meet the managed Software Engineering Institute Level 4 software development process in use by the architecture and integration contractor. The TCC allows integrated access to a user-configurable suite of software development tools. These tools span the software development life cycle from analysis through test.
WebRT	Automated risk tracking, action-item tracking, and defect tracking are handled using the freely available open source WebRT tool <www.bestpractical.com/index.html>. WebRT 1.0.1-4 has been customized to provide a Web-enabled tool to track and manage defects, issues, risks, and action items within OneSAF.
Java	Java provides a platform-independent development language and development kit to OneSAF. Sun's Java version 2.0 <www.javasoft.com>, along with the Software Development Kit (SDK) version 1.4.1, provides the Java language programming foundation for the OneSAF integrated drive electronics. OneSAF is reviewing the capabilities and schedule for stepping up the next major release of the Java SDK.
XML Spy	As data architecture and management play a critical role across the pre-exercise, run time, and post-exercise activities, OneSAF is leveraging eXtensible Markup Language (XML) technologies including XML Spy <www.xmlspy.com>. XML Spy version 4.0 provides the OneSAF users within the IDE the ability to create XML schemas that comply with the OneSAF Data Interchange Formats (DIF) standards. XML Spy features a format checking and validation tool to cross-check a document against its DIF.

success of ModSAF. For OneSAF these tools leverage Web- and e-mail-based technologies. For a list and description of these tools and technologies, see Table 1.

These tools are actively in use and have provided huge dividends in terms of user engagement and feedback. As part of the normal OneSAF development process, users review demonstrable capabilities and code and then electronically submit their comments, enhancements, and/or changes with supporting documentation to the Web-served comment and defect repository. These submissions are reviewed, categorized, and assigned for action.

After OneSAF Full Operational Capability at the end of fiscal year 2005, these Web-based tools may be enhanced to support code updates that can be inserted into an integration branch, compiled, and automatically regression tested with the results posted to the OneSAF Web and notification e-mailed to the developer. Currently, architecture compliance tools and processes exist to allow external developers to plan for a specific level of integration with the OneSAF code. The external developers' decisions are dependent on their requirements and investment in existing applications. Prior to formal baseline integration, new code that has been successfully integrated will be posted for download at the *users own risk*.

A formal OneSAF Configuration Control Board (CCB) will choose which newly integrated capabilities to incorporate into the next formally released baseline. Once incorporated into the baseline, PM OneSAF assumes responsibility for these enhancements, distributes them within the normal baseline distribution process, and removes the pre-baselined code from the use-at-your-own-risk Web page.

In addition to sponsoring CCB meetings, OneSAF now holds regular user group meetings for both the domestic and international M&S communities. This user group meeting gives users the opportunity to exchange relevant information about OneSAF and its individual programs, demonstrate new capabilities, voice concerns, raise issues, and make recommendations.

Filling the Gaps
It was also critical to OneSAF's success to improve several ModSAF architectural blemishes. These critical improvements include (1) providing a more composable and extensible software architecture; (2) focusing on and providing tools for non-programmers to extend the list of simulated entities, units, and behaviors; (3) providing mechanisms to support greater success when integrating user-developed code; and

(4) providing mechanisms to fully document the interfaces and code delivered.

To meet the challenges and limitations of earlier simulation systems, the OneSAF architects applied a software-based product-line architecture development approach. The product-line approach concentrates on identifying and defining interfaces between independent, architecture-level components, and then specifies how these existing components can be automatically composed into useful end-user applications. Looking back on the early – circa 2001– OneSAF architectural development, three key architecture-related tenets stand out as significant enablers to the four improvement areas mentioned above, and to the overall success of the program. These tenets include a coordinated architectural vision, an iterative and incremental development process, and close user and developer collaboration.

Tenet I: Create a Coordinated System Architectural Vision
For OneSAF, this was essential in orienting and maintaining forward momentum on two of the four critical improvement areas: overall architecture support to composability and extensibility requirements, and support for tool-based extensibility.

Composability and extensibility were viewed as essential for OneSAF because these characteristics were especially limited by ModSAF's architecture. ModSAF's aggregate applications made it difficult, expensive, and time consuming for the community to make specific independent modifications and for these modifications to be integrated back into the baseline. This was particularly true when multiple modifications were made to a single application.

The OneSAF architecture vision was developed early on in concert with the procurement team, the users, and the developers. The vision focused on system-level composability and the architecture's ability to support independent component development along well-defined interfaces within quality and functional compliance specifications. This was the key to enabling the government's task order procurement strategy of issuing multiple contracts to develop the independent system components.

An architecture and integration contract was also issued to finalize the interface, functional, and quality specifications, as well as to create the development infrastructure [3] and develop the initial toolset. By forging this vision early on, the program was able to develop a set of objective measures called *interface maturity levels* used to define objectives and emphasize and measure progress against the OneSAF requirement set.

Since program inception, from the

architecture level down to implementation, particular emphasis has been on the composability tools allowing end-users to compose new entities, units, and their associated behaviors from existing software primitives without the need to write or even access the source code. These tools are highlighted in the OneSAF architecture as the Model Composer tools and include the Entity Composer, the Unit Composer, and the Behavior Composer.

The Entity Composer allows a OneSAF simulation entity such as an aircraft, helicopter, tank, truck, or individual combatant to be composed from individual model components using a Windows-based graphical user interface (GUI). Model components may include visual or other radar-based sensors, specific types of weapons, specific communications devices, and specific mobility components such as *wheeled* or *tracked*. Additionally, the user can select specific types of physical models that regulate the vulnerability, load carrying capacity, and other physical aspects of the entity.

The Unit Composer supports grouping individual entities into military units and civilian organizations using a GUI-based front end. The Unit Composer allows visualization and modification of all entities and previously constructed units. Once a unit is constructed, specific behaviors defining the doctrine and tactics of the unit can be associated with the unit. These behaviors are constructed using Behavior Composer.

The Behavior Composer allows the graphical construction of entity and unit behaviors from existing primitives (software coded behaviors) and other composite behaviors. Composite behaviors are simply behaviors that are made up of other composite or primitive behaviors. The Behavior Composer allows the specification of sequential or parallel behaviors that provide the automated control, reactions, and overall behaviors of entities and units.

Tenet II: Use an Iterative and Incremental Development Process
For OneSAF, an iterative and incremental process enabled two important effects. First, it allowed the program to create and test, through successive iterations, a set of consistent, comprehensive, and useful architectural- and implementation-level documentation. It also allowed the program to test and streamline its support for external development and integration, again, by applying lessons learned through successive iterations of the integration and test process. Specifying the architectural vision on paper and then working toward that vision in code allowed negotiated changes to the architecture where necessary to maintain consisten-

cy with the code. Changes occurred due to developmental breakthroughs, clearer understanding of requirements, or necessary changes due to reuse of legacy applications.

OneSAF continues to use an eight-week build process as the cornerstone for software and system engineering activities. The builds begin with defined, measurable, and in many cases, demonstrable objectives and then progress through analysis, design, code, and unit test. During the subsequent build, the previous build's products are sent through a system-level integration and test process. At yearly intervals, the system is packaged and delivered to a restricted set of beta-test sites. OneSAF is currently in its Block C development cycle and has successfully released its Block A and B system products to more than 50 beta-test sites.

For OneSAF, the iterative and incremental process also enabled the drive to demonstrate capabilities early and often to support a rich level of user interaction and feedback across the ACR, RDA, and TEMO domains.

Tenet III: Establish and Maintain Close User and Developer Collaboration

For OneSAF, this enabled continual feedback, prioritization, and interpretation of key user needs and requirements. The current success of the OneSAF program is also attributed to these close developer and user interactions. From the start, the development and user representative teams have been collocated in the OneSAF Integrated Development Environment, thereby enabling easy and frequent interaction between the more than 100 system and software engineers, the government management and technical personnel, and the domain (ACR, RDA, and TEMO) user representatives. This interaction is key to shortening the development cycle, as domain-specific questions can be quickly resolved.

Conclusion

From the start, OneSAF leadership has encouraged non-traditional software development methods. Open source development remains central as a new way to create, distribute, proliferate, and extend the simulation capabilities promised by OneSAF. Using lessons from earlier simulations, open source concepts played heavily in the selection and creation of processes and tools to meet the requirements of the OneSAF program.

Open source development has already paid large dividends to the OneSAF program ranging from enhanced Web-based communications between developers and users gained from using open source Web-based servers, mail lists, and request tracking

software, to the advanced native capabilities within the Java development environment. OneSAF leadership believes the benefits of open source development will expand upon OneSAF's formal release by allowing a wide range of developers to contribute to and propose extensions back to the OneSAF baseline. Once integrated into the core baseline, a capability developed by a given organization is available for use and extension by the community of OneSAF users.

Although OneSAF is still in development, once formally released in fiscal year 2006, it will be available and distributed with source code free of charge to any organization within the DoD with a valid OneSAF requirement. While OneSAF is focused as an Army/DoD program, other inter-agency organizations (e.g., other services, homeland defense, emergency response groups/police, etc.), industry, and academia can gain access to the application.

OneSAF will also be available to the international community. Typically, international arrangements are made either through data exchange agreements, foreign military sales cases, or project agreements. Because of the sensitive nature of OneSAF, removing the non-exportable data and algorithms will develop a specialized international baseline.

Finally, as OneSAF prepares to enter its final year of development prior to formal release, it will evaluate and grow the necessary capabilities to accommodate distributed open source development. It is expected, based on lessons learned and experience to date, that the necessary tool and process changes will be small incremental changes versus an explosive big bang event.

To find out more about the OOS, please contact the authors, PM OneSAF, Lt. Col. John "Buck" Surdu at <john.surdu@peostri.army.mil>, or visit <www.onesaf.org>.◆

References

1. Hasan, Ragib. "History of Linux." Vers. 2.1. University of Illinois at Urbana-Champaign, July 2002 <https://netfiles.uiuc.edu/rhasan/linux>.
2. Training and Doctrine Command. "Mission Needs Statement." U.S. Army, 2004 <http://onesaf.org/1SAFMNS.doc>.
3. Wittman, Robert L., and Anthony J. Courtemanche. "The OneSAF Product Line Architecture: An Overview of the Products and Process." SimTeCT 2002. Melbourne Convention Center, Melbourne, Australia, 13-16 May 2002.
4. Webbink, Mark H. "Understanding Open Source Software." Society for Computers and the Law (51) Mar. 2003 <www.nswscl.org.au/journal/51/Mark_H_Webbink.html>.

About the Authors

Douglas J. Parsons is the lead engineer of the Intelligent Simulation Systems Team at the U.S. Army Program Executive Office for Simulation, Training, and Instrumentation. His primary focus is toward the successful development of the One Semi-Automated Forces Objective System. Parsons has a Bachelor of Science in mechanical engineering from North Dakota State University, a Master of Science in systems management from Florida Institute of Technology, and a Master of Science in industrial engineering from the University of Central Florida.

Program Executive Office-Simulation Training and Instrumentation (PEO-STRI)
12350 Research PKWY
Orlando, FL 32826
Phone: (407) 384-3821
E-mail: doug.parsons@peostri.army.mil

Robert L. Wittman Jr., Ph.D., currently works for the MITRE Corporation supporting the One Semi-Automated Forces Objective System program. He has been part of the U.S. Department of Defense modeling and simulation community since 1990. He has a Bachelor of Science in computer science from Washington State University, a Master of Science in software engineering from the University of West Florida, and a doctorate in industrial engineering from the University of Central Florida.

MITRE Corporation
3504 Lake Lynda DR
Orlando, FL 32817
Phone: (321) 235-7601
E-mail: rwittman@mitre.org

Introduction to the
User Interface Markup Language

Jonathan E. Shuster
Acumenia, Inc.

Current languages and tools for creating software user interfaces are tightly tied to the computing device on which the user interface runs. For example, development teams often use Java or C++ for graphic user interfaces, Hyper Text Markup Language for Web interfaces, the Wireless Markup Language for cell phones, and VoiceXML [eXtensible Markup Language] for voice interfaces. The tight coupling of language to device means that to use a variety of devices with software systems, development teams must master different languages and toolkits and maintain different code bases for each device. This article introduces the User Interface Markup Language (UIML), an open XML-compliant language capable of describing user interfaces for virtually any computing device. It describes how UIML can be used for creating multi-platform user interfaces, how it is being applied in defense applications, and introduces UIML syntax.

Let us take a look at two different scenarios of development teams challenged to integrate different computing devices or upgrades into their systems.

Scenario 1: A development team is charged with creating a software system that users can access through a variety of client computing devices. Tasked with providing desktop access for internal users, Web access for external users, access via a wireless-enabled Personal Digital Assistant (PDA), and voice-only access through telephones, the development team writes user interfaces in Java for the desktop platform, Hyper Text Markup Language (HTML) for the Web, and VoiceXML [eXtensible Markup Language] for the voice interface. Having selected Palm devices, they write the PDA user interface in C for the PalmOS. As the system evolves, they spend considerable effort making the same or similar changes to each of these user interfaces. A year into the project, management decides to drop the Palm device and instead support PocketPC PDAs. The team rewrites the Palm user interface to run on PocketPCs, which increases the project cost and delays the schedule.

Scenario 2: A weapon systems project is charged with improving the usability characteristics of its software user interfaces and adopts an iterative usability design process. The user interface team needs to get an early start on creating usability prototypes, but the deployment hardware, operating system, language, and user interface toolkit have not been selected yet. The user interface design team creates the usability prototypes in VisualBasic, and when the deployment platform is selected, rewrites the entire user interface in C++. A few years later, a *technology refresh* is planned to upgrade the deployment platform to take advantage of new technologies. Plans for the upgrade are dropped because the expense of rewriting the user interface for the new platform is prohibitive.

Unfortunately, scenarios like these two are all too common for development teams trying to integrate different computing devices into their systems. While hardware vendors have given us a rich array of computing devices – PDAs with wireless connectivity, cell phones, even the telephone – the promise of these devices in providing portable and easy-to-use access to data is difficult to realize using conventional software development tools.

The problem is that the languages and toolkits we use to describe software user interfaces are tightly tied to the underlying platform. Not only does this require development teams to maintain proficiency in many different languages, but it is difficult to achieve reuse across these languages: When the user interface changes, changes must be applied separately to each platform's user interface code.

What if a single language existed that could describe user interfaces independently of client device? Such a language would need to be able to completely describe the user interface and its interactions with the underlying application logic. It would need to be flexible enough to describe user interfaces using widely different metaphors such as graphic user interfaces, voice interfaces, interfaces for automotive on-board computers with unconventional interaction devices, and interfaces for devices not yet invented such as those embedded in soldiers' uniforms.

Such a language exists: the *User Interface Markup Language* (UIML) [1]. UIML is an XML-compliant language created with the goal of describing any user interface for any device, regardless of operating system or target programming language. User interface descriptions written in UIML are *rendered* for specific target platforms, much in the same way that documents described in HTML are translated into viewable documents by Web browsers. UIML renderers can either be interpreters that read the UIML and create the user interface at run time, or compilers that translate UIML into other languages.

UIML renderers have been developed for Java, HTML, Wireless Markup Language (WML), VoiceXML[1], .NET [2], Python [3], and even for augmented reality applications [4]. UIML was the subject of an international conference in 2001[2].

Original Motivation for UIML

UIML was developed by a team of researchers in Blacksburg, Va., starting in 1997, and has been enhanced by several organizations, including Virginia Tech. The team was frustrated with the poor usability characteristics of many software user interfaces and the difficulty in creating good user interfaces with existing languages and tools. Increasingly, user interface design required skills often not present in development teams, such as visual layout, an orientation toward how human users carry out tasks, and graphic design. Yet, designers were required to use programming languages such as C, C++, and Java, which were fundamentally designed to describe application logic.

These problems led to a desire to create a language designed specifically for user interface design. To stay oriented to the needs of user interface designers, UIML was designed as a declarative language; that is, UIML would describe what the user interface looks like (as HTML describes documents), rather than the steps followed in building the user interface (as do languages such as C++ and Java). UIML is an XML-compliant language taking advantage of the availability of XML tools. UIML is an open language being standardized through the Organization for the Advancement of Structured Information Standards (OASIS) [5][3]; the language specification is available at <www.uiml.org>.

UIML Applications in DoD

UIML is gaining interest in the Department of Defense as a technology for implementing user interfaces for complex software systems with long lifetimes. The Navy's Tactical

Tomahawk Weapons Control System (TTWCS) program sees UIML as a way to help automate the generation of deployable code from usability prototypes. UIML also addresses the need to adapt weapon system control user interfaces to accommodate different watchstations used on different ship classes. TTWCS is sponsoring the development of UIML authoring and deployment tools under the Small Business Innovation Research (SBIR) program. A preliminary estimate made under the SBIR Phase I project suggested that adopting UIML could save the program between $1.5 million and $3 million for a typical TTWCS software version, allowing accelerated delivery of critically needed new features to the fleet.

The Navy's DD(X) shipbuilding program sees UIML as an excellent way to implement a common computing environment across all shipboard software systems. UIML makes it possible to apply common characteristics to all user interfaces such as the *look and feel* and layout of interaction mechanisms. UIML also provides a way to achieve significant reuse across software systems, not only for visual characteristics, but also for underlying mechanisms such as the programming interfaces used to interact with the underlying applications.

In the Army, UIML has been used on the Army Training Information Architecture program to make it possible to deliver training documentation to small-aperture devices such as handheld computers and PDAs. Even though much of the Army's training documentation is in HTML format, viewing legacy HTML on PDAs presents a host of usability problems (described as "like watching TV through a soda straw"). In a pilot project, legacy HTML was converted into UIML, and then delivered by a UIML server to the client device. Based on the device requesting the document (desktop or PDA), the UIML server transformed the UIML description based on device characteristics, then rendered the UIML into HTML tailored for optimal viewing on the device.

UIML: A Canonical Meta-Language

UIML is a canonical meta-language for describing software user interfaces. As a canonical language, UIML regularizes the idiosyncrasies in syntax across device languages. The table below shows how UIML reduces the syntactical differences across HTML, Java, and C++ with the GTK+ (Gnu's Not Unix [GNU] Image Manipulation Program Toolkit) widget set to canonical form.

```
HTML:
    <input name="submit">
Java:
    JButton submit = new JButton();
C++/GTK:
    GtkWidget *button = gtk_button_new();
UIML:
    <part class="Button" id="okButton"/>
```

By reducing the definition to a standard form, renderers can be used to translate UIML into any of the other forms. This means that as new devices, operating systems, and programming languages emerge, it is not necessary to change UIML user interface descriptions. Rather, the UIML is simply re-rendered for the new platform.

Being a meta-language gives UIML the flexibility to describe user interfaces for widely different devices. Rather than having many toolkit-specific tags (such as <menu> or <button>) covering all possible user interface metaphors, UIML uses a few powerful tags (such as <part> and <property>). As with XML, where you add a schema to make it useful, you add a *vocabulary* to UIML to define the abstractions that are needed to describe the user interface. These abstrac-

Figure 1: *UIML Examples 1-4*

UIML Examples 1-4

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC "-//UIT//DTD UIML 3.0x Draft//EN"
"http://uiml.org/dtds/UIML3_ 0a.dtd">

<!-- Example 1: The UIML Skeleton -->

<uiml>
    <interface>
        <structure>...</structure>
        <style>...</style>
        <content>...</content>
        <behavior>...</behavior>
    </interface>
    <peers>
        <logic>...</logic>
        <presentation base="..."...">
    </peers>
</uiml>

<!-- Example 2: Defining Parts -->
<structure>
    ...
    <part id="okButton" class="Button"/>
    ...
    <part id="aPanel" class="Panel">
        <part id="aField" class="Field"/>
    </part>
    ...
</structure>

<!-- Example 3: Defining Properties of Parts -->
<style>
    ...
    <property class-name="Button" name="size">20,20</property>
    ...
    <property part-name="okButton" name="size">40,20</property>
    ...
</style>

<!-- Example 4: Two Ways to Define Content -->
<style>
    ...
    <property part-name="okButton" name="text">Okay</property>
    ...
    <property part-name="aField" name="text">
        <reference constant-name="welcomeText"/>
    </property>
    ...
</style>
```

tions can be platform-specific (e.g., defining a JButton class for Java Swing), or generic across similar platforms (e.g., a *Button* class to use for graphic user interfaces)[4]. Vocabularies can be used to define domain-specific abstractions such as *SteeringWheelButton* for automotive user interfaces.

UIML vocabularies define allowable parts and classes, properties, and events, and map these abstractions to specific widgets in the target language and toolkit. For example, a *Button* class can be defined that maps to *java.swing.JButton* for Java with the Swing toolkit. Vocabularies have been defined for Java, HTML, VoiceXML, WML, and other target languages[5].

What Does UIML Look Like?
The UIML Skeleton
Describing a user interface requires answering six questions:
1. What structure of parts makes up the user interface?
2. What presentation style should be used for each part?
3. What is each part's content?
4. What behavior do parts have (that is, what should happen when, for example, a user clicks on a button)?
5. How does the user interface connect to the underlying application logic?
6. How are parts mapped to widgets in the target toolkit?

UIML separately describes these six aspects of the user interface definition. The answers to the first four questions define the interface itself; the last two define how the interface interacts with the outside world. Thus, the basic skeleton of a UIML user interface is shown in Example 1 in Figure 1.

The first group of lines is an XML document type declaration that marks this as a UIML document. The remaining lines show the basic skeleton of a UIML document. Note that the <structure>, <style>, <content>, and <behavior> tags address the first four questions about the user interface. The <logic> tag addresses connections to the underlying application logic (question No. 5), and the <presentation> tag addresses toolkit mappings (question No. 6).

Defining these six aspects separately enables reuse. For example, consider an automotive manufacturer creating Web versions of the owner's manuals for each of its models. It is not unusual for owner's manuals to be translated into as many as 25 different human languages depending on where the model is sold. Using HTML, 25 separate Web applications would be needed for each model. If the structure of the owner's manual changes, the changes would need to be applied to all 25 Web applications.

With UIML, the owner's manual applica-

tion would be defined as a single UIML document. Different <content> sections would be defined for each language, and the appropriate content section specified at rendering time. The structure, style, and other characteristics of the owner's manual application are defined only once, and changes to these characteristics need only be applied in one place.

Similarly, reuse can be achieved with the other major sections of a UIML document. For example, different style guidelines can be applied to user interfaces by using different <style> sections. Application interfaces, defined in the <logic> section, can be written once and reused in UIML written for different platforms.

UIML has several mechanisms to support reuse. Most notably, it includes the concept of *templates*, external files containing commonly used UIML definitions. In addition, some renderers allow specifying UIML tags by name; for example, allowing multiple <content> tags for different human lan-

Figure 2: *UIML Examples 5-8*

```
UIML Examples 5-8

<!-- Example 5: Defining Alternative Content Sets -->
<content id="English" xml:lang="en-US">
    <constant id="welcomeText">Welcome</constant>
    …
</content>
<content id="French" xml:lang="fr">
    <constant id="welcomeText">Bienvenue</constant>
    …
</content>

<!-- Example 6: Defining Behavior -->
<behavior>
    <rule>
        <condition>
                <event class="actionPerformed" part-name="okButton"/>
        </condition>
        <action>
                <property part-name="aWindow" name="visible">
                        FALSE
                </property>
                <property part-name="aDialog" name="visible">
                        TRUE
                </property>
        </action>
    </rule>
</behavior>

<!-- Example 7: Making Application Calls -->
<behavior>
    <rule>
        <condition>
                <event class="actionPerformed" part-name="okButton"/>
        </condition>
        <action>
                <property part-name="aLabel" name="text">
                        <call name="Counter.count"/>
                </property>
        </action>
    </rule>
</behavior>

<!-- Example 8: Mapping UIML Calls to The Application -->
<logic>
    <d-component id="Counter" maps-to="AppCounter">
        <d-method id="count" return-type="int" maps-to="bumpCount"/>
    </d-component>
</logic>
```

guages, and selecting which content set to use at rendering time.

The following sections give an overview of the six sections of a UIML document. It is not possible to completely describe UIML syntax in one article; however, considerable information about UIML is available in the references listed at the end of this article.

Defining Structure

The <structure> tag defines the parts that make up the user interface. Nested parts are defined, appropriately, by nesting <part> tags. Example 2 in Figure 1 (see page 16) shows UIML defining a top-level part (a button of class *Button* named *okButton*), and a set of nested parts (a panel containing a text field).

Defining Style

The <style> tag describes the properties of each part. Properties can be associated with either individual parts or classes of parts, as shown in Example 3 in Figure 1 (see page 16).

In the first <property> tag, the default

size of all buttons in the Button class is set to 20 by 20 pixels. In the second <property> tag, the size of the button named okButton is set to 40 by 20 pixels.

Defining Content

Content can be defined in UIML as a part's property, or can be defined in a separate content section as described earlier. Example 4 in Figure 1 (see page 16) shows both methods: The first property tag defines the content of okButton as the text *Okay*. The second property tag references a constant named *welcomeText*.

The constant referenced in the second property tag is defined in the <content> tag. Example 5 in Figure 2 (see page 17) shows UIML defining two alternative content tags, for English and French, for selection at rendering time.

Defining Behavior

Behavior is defined as a set of rules. Each rule describes an action to be carried out under a given condition. Actions can include changing properties or making application

calls. Example 6 in Figure 2 (see page 17) shows a rule specifying that when a button is pressed, the active window is closed and a confirmation dialog is opened. The window is closed by setting its *visible* property to FALSE; similarly, the dialog is opened by setting its *visible* property to TRUE.

In this example, the condition is the occurrence of an event. Allowable event types are defined in the vocabulary. Other conditions may also be used such as equality between a property and a constant, for example.

Making Application Calls

Calls to the underlying application are defined with the <call> tag. In Example 7 in Figure 2 (see page 17), the result of the call is used to set the content of a text label.

The <call> tag references the value returned by the *count* method on the *Counter* object. Placing the call tag in the <property> tag has the effect of resetting the text property to the value returned by the <call>.

Mapping Calls to Application Logic

Note that <call> tags define application calls in an abstract form. The <logic> section maps UIML calls to specific objects and methods (or procedures) in the underlying application. This means calls can be easily remapped to different application interface calls simply by changing the definition in the <logic> section. Example 8 in Figure 2 (see page 17) maps the abstract call *Counter.count* to a specific object and method in the underlying application (*AppCounter.bumpCount*).

In this example, the <d-component> tag (for *defined component*) maps the UIML component *Counter* with the application's *AppCounter* object. Similarly, the <d-method> tag (for *defined method*) maps the UIML *count* method with the AppCounter object's *bumpCount* method, and specifies an integer return type.
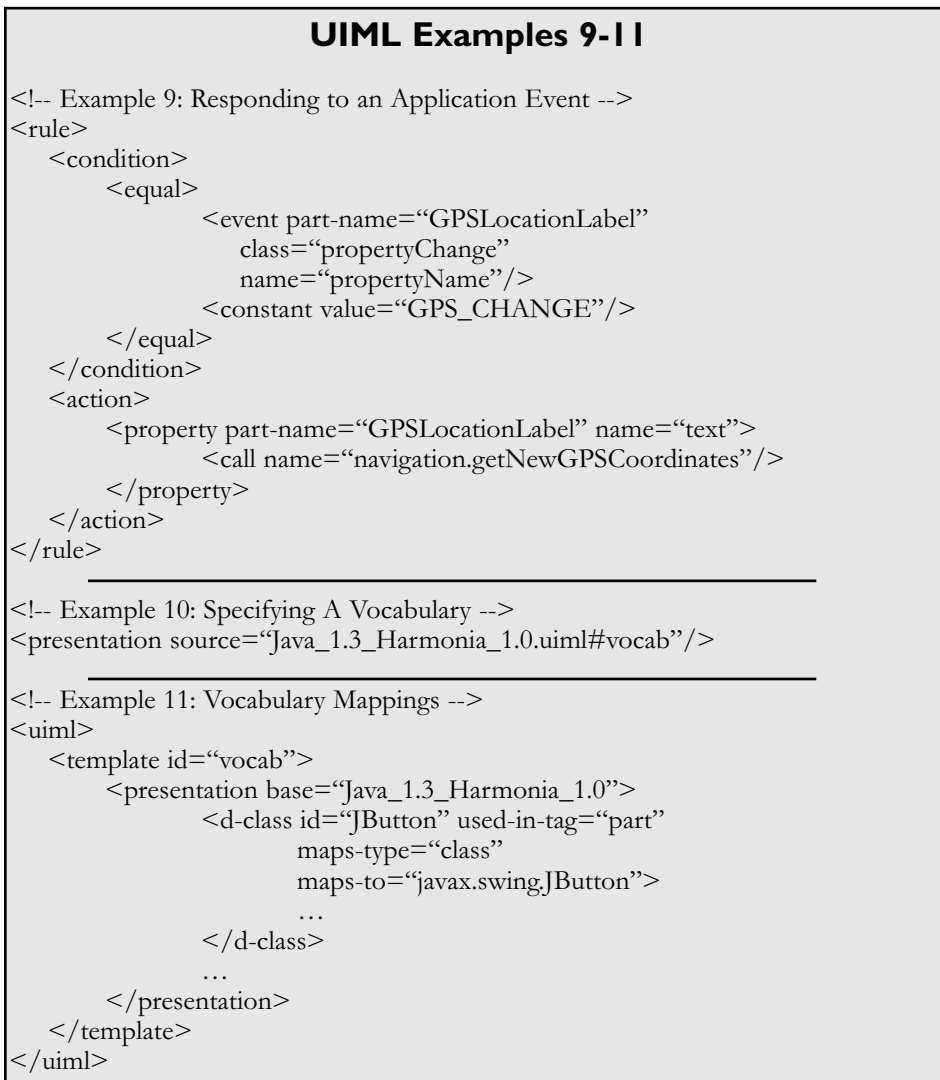
Responding to Application Events

Rules can be defined that allow the user interface to respond to events received by the underlying application. Example 9 in Figure 3 shows a rule that allows a part to display updated global positioning system (GPS) coordinates upon receiving an event indicating the location has changed.

In this example, the condition that fires the rule is when the event equals a certain constant. The action is to call a method to get new GPS coordinates, and display the return in the GPSLocationLabel part.

Mapping UIML Abstractions to Specific Toolkit Widgets

The <presentation> section defines the vocabulary to be used. Normally, the <pre-

Figure 3: *UIML Examples 9-11*

```
UIML Examples 9-11

<!-- Example 9: Responding to an Application Event -->
<rule>
    <condition>
        <equal>
                <event part-name="GPSLocationLabel"
                    class="propertyChange"
                    name="propertyName"/>
                <constant value="GPS_CHANGE"/>
        </equal>
    </condition>
    <action>
        <property part-name="GPSLocationLabel" name="text">
                <call name="navigation.getNewGPSCoordinates"/>
        </property>
    </action>
</rule>

<!-- Example 10: Specifying A Vocabulary -->
<presentation source="Java_1.3_Harmonia_1.0.uiml#vocab"/>

<!-- Example 11: Vocabulary Mappings -->
<uiml>
    <template id="vocab">
        <presentation base="Java_1.3_Harmonia_1.0">
                <d-class id="JButton" used-in-tag="part"
                    maps-type="class"
                    maps-to="javax.swing.JButton">
                    …
                </d-class>
                …
        </presentation>
    </template>
</uiml>
```

Introduction to the User Interface Markup Language

sentation> tag references a vocabulary defined in an external file, as shown in Example 10 in Figure 3.

The vocabulary itself maps abstractions used in the UIML user interface to specific toolkit widgets. Example 11 in Figure 3 shows part of a Java vocabulary that maps the JButton class to a specific Java Swing object.

Conclusion

In the early days of personal computing, peripheral devices were tightly coupled to application software. This meant that users had to make sure the software they bought was compatible with their specific printer, modem, or other peripherals. Making device drivers a part of the operating system uncoupled peripherals from applications, and now users only need to worry about buying software and peripherals that are compatible with their operating system. This was a tremendous step forward in the evolution of personal computing.

UIML can have a similar impact on application development. By defining user interfaces in a platform-independent manner, UIML decouples the user interface from the underlying computing device. This makes it easier to use a wide range of computing devices in software applications, and results in user interfaces that are much more easily adapted to new computing devices as they emerge on the market.◆

References

1. Abrams, M., C. Phanouriou, A.L. Batongbacal, S. Williams, and J.E. Shuster. UIML: An Appliance-Independent XML User Interface Language. Proc. of the Eighth International World Wide Web Conference, May 1999 <www8.org/w8-papers/5b-hypertext-media/uiml/uiml.html>.
2. Luyten, K. "UIML.Net: A UIML Renderer for .Net." Limburgs Universitair Centrum, Jan. 2004 <http://research.edm.luc.ac.be/kris/projects/uiml.net>.
3. Cherkashin, E. "Python UIML Renderer." Apr. 2001 <http://freshmeat.net/projects/pyuiml>.
4. Sandor, C., and T. Reicher. CUIML: A Language for Generating Multimodal Human Computer Interfaces. Proc. of the UIML 2001 Conference, May 2001 <www.uiml.org/cd_updates/UIML_2001_Conference/papers/Sandor_paperFinal.pdf >.
5. Abrams, M., and J.W. Helms. "User Interface Markup Language (UIML) Specification 3.1." Working Draft 3.1. OASIS Open, Inc., 2004 <www.oasis-open.org/committees/tc_home.php?wg_abbrev=uiml>.

Notes

1. Tools for using UIML have been developed by a number of organizations, most notably Harmonia, Inc., of Blacksburg, Va., <www.harmonia.com>. The U.S. Navy is sponsoring the development of additional tools through its Small Business Innovation Research program, including the development of a UIML authoring environment.
2. Information about this conference, including papers presented, is available on the Web at <www.aristote.asso.fr/sem/sem0101UIML-en. html>.
3. For more information about the OASIS UIML standardization technical committee, and for the most recent draft UIML specification, see <www.oasis-open.org/committees/uiml>.
4. UIML is *object-based* in that it allows defining classes of parts, but does not support other *object-oriented* concepts such as inheritance.
5. The UIML Web site, <www. uiml.org>, is a good site for information on UIML. Besides specifications and document type definitions, a number of UIML vocabularies are posted here (see <www.uiml.org/toolkits/index.htm>).

About the Author

Jonathan E. Shuster, founder and president of Acumenia, Inc., provides management and technical services to software engineering organizations. He has led development teams for Navy, Army, and Department of Energy applications ranging from information systems to simulation models to three-dimensional stereo-immersive virtual environments. He was a member of the original team that invented the User Interface Markup Language, an eXtensible Markup Language-compliant language for creating user interfaces for virtually any computing platform. His passion is helping people understand information-related problems and deploying the appropriate technology to solve those problems

Acumenia, Inc.
1872 Pratt DR, STE 1425
Blacksburg, VA 24060
Phone: (540) 250-1300
Fax: (724) 271-0025
E-mail: jshuster@acumenia.com

DO-178B Certified Software:
A Formal Reuse Analysis Approach

Hoyt Lougee
Foliage Software Systems

In these lean economic times, avionics manufacturers have a heightened interest in certifiable software reuse as an alternative to developing design-from-scratch software for next-generation systems. When appropriate, software reuse can provide significant return on investment and time-to-market advantages.

As indicated in the December 2004 CROSSTALK article "Reuse and DO-178B Certified Software: Beginning With the Basics," [1] reuse is defined simply as "using previously existing software artifacts." These artifacts may or may not have been designed for reuseability. In fact, these artifacts may be proposed artifacts intended for future reuseability.

Artifacts include all products of a certification development process: planning data, requirements data, design data, source code, configuration management records, quality assurance records, and verification data. Artifacts extend over all functional areas of software. A sample breakdown is illustrated in Figure 1.

Reuse Stakeholders

In identifying the purpose and goals, and selecting the optimal approach, the reuse analysis must take into account the views of the various stakeholders. Typical stakeholders are described in Table 1. Each of these stakeholder groups normally has different expectations for reuse, as well as different insight into the pros and cons of a particular approach.

First-tier sources[1], for example, are concerned with cost, schedule, functionality, and safety. The reuse strategy particulars that provide these benefits are typically of little importance to these first-tier sources. Senior management usually represents the interests of the external customer stakeholders.

On the other hand, cost and schedule are not of paramount importance to the Federal Aviation Administration/ Designated Engineering Representative (FAA/DER). Safety is the primary driver of the certification authorities (CA). The reuse strategy selected, however, can make the CA's job easy or complex. A properly designed and executed reuse strategy will make it easier for the FAA/DER to trace verification among configurations to provide more confidence and reduce the amount of review to be performed.

Although cost, schedule, functionality, and safety are also important to senior management, the long-term efficiency and profitability of their product lines are also a priority, as are the competing short-term budget limitations. Senior management drives reuse strategies that accommodate the *big picture*.

Cost, schedule, functionality, safety, and long-term efficiency and profitability goals are flowed to project management. At this level, however, concerns about short-term feasibility, effects on staffing, and the lower-level tactical implementation gain importance.
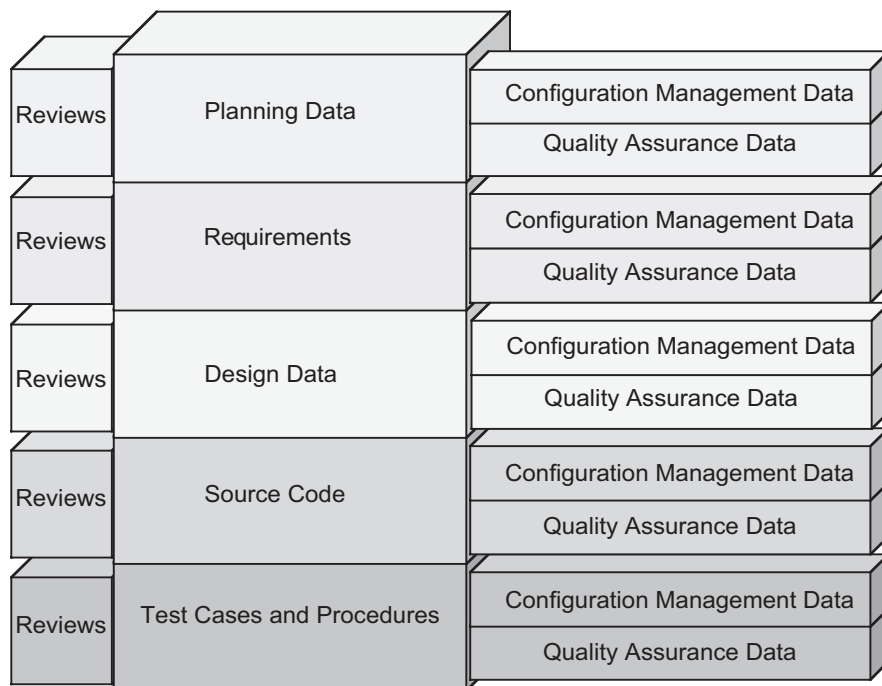
System engineers drive the hardware/software functionality of the system and are often the source of system-level requirements allocated to software. Since many software strategies must be supported by or aligned to candidate hardware configurations, systems engineers are critical participants in the reuse analysis process. Cost, schedule, functionality, safety, long-term efficiency and profitability goals drive the systems engineer across both the hardware and software domains.

Development engineers and test engineers have the cost, schedule, functionality, safety, long-term efficiency and profitability goals, as well as the expectation for creating good software: software that is flexible, extensible, and sustainable. The short-term feasibility and the actual *nuts and bolts* of the implementation are also placed on their doorstep.

Quality engineers must verify the software configurations produced and must be efficient when doing so. Efficient testability is a challenge for the quality engineer in two ways:

1. Reusing tests and test results for reuseable components can reduce the amount of testing performed on new platforms, as well as reduce the overall number of tests to be created.
2. Ensuring that end-to-end functionality is verified to make certain that reused components are used correctly.

Figure 1: *Reusable Artifacts*

Clearly, these two goals can contradict one another. If reusable components are verified at the component level, end-to-end verification testing may redo much of the testing. Moreover, the FAA/DER typically insists that significant end-to-end testing be performed to ensure all software components integrate properly. Obviously, compromises are in order.

Finally, the configuration managers have the considerable task of configuring the reuseable elements to allow controlled, traceable software releases. This task must be performed early in the process or the reuse effort suffers accordingly. The viability of the reuse strategy depends upon the ability to configure the elements appropriately. This ability depends upon a coherent and reasonable configuration strategy, a suitable toolset, and appropriate training and staffing.

Reuse Analysis and Incorporation Process

As discussed in [1], reuse types vary according to the reasons for reuse, as well as the character of the elements to be reused; however, the reuse analysis and incorporation process is the same. Figure 2 describes a standard reuse analysis and incorporation process.

Clearly Identify Purpose and Goals

The first step in our reuse analysis and incorporation process is to clearly identify the purpose and goals of the reuse effort. The reuse purpose is the business problem to be addressed. Reuse goals, on the other hand, are the competing benefits that different reuse strategies yield.

To effectively identify purpose and goals, you must rid yourself of the idea that you can have everything: Tradeoffs must be made. Optimal purpose identification often entails initial purposes that grow, shrink, divide, or combine as the goals are examined. Therefore, involving higher levels of management is often crucial in the give and take of the purpose and goal identification.

Further tradeoffs must be made in prioritizing goals. The relative importance of potential reuse benefits will dictate the degree to which they will be considered in selecting the reuse strategy. That is not to say, however, that one benefit will completely exclude any of the other benefits; the identification of purpose and goals is based upon a relative prioritization of these factors.

Potential short-term and long-term benefits must also be addressed when identifying goals and purpose. For example, flexibility, extensibility, and sustainability typically correlate with increased long-term benefits, higher short-term cost, and longer schedule. To lower short-term cost and shorten schedules, these factors might be given lower priority.

The identification of purpose and goals entails the five steps detailed below.

Identify Purpose

As indicated above, the reuse purpose is the business problem to be addressed. The purpose is the project description over which the reuse effort will be applied and describes the scope and domain of the effort. For example, the need to introduce a new product line is a reuse purpose. Another purpose might entail creating a family of controllers – the family representing a single *project* over which reuse analysis will be performed. Yet another purpose could include functional modifications necessary to enhance an existing product line.

Note that these purposes do not and must not relate to the reuse strategy. An example of a poorly defined purpose would be "to create a common reuseable library." This purpose presupposes the results of the analysis: a reuseable library. The *correct* strategy has already been decided before the analysis has been performed. You must clearly define your purpose to allow bounding of the problem to be addressed.

Identify and Quantify Goals

Next, you must determine the relative importance of potential reuse benefits, especially in terms of long-term versus short-term benefits.

To identify goals, you must catalog (list and describe) the expected potential reuse benefits. A variety of competing reuse benefits typically present themselves. For

Stakeholder	Internal/External
Airframers	External
First-Tier Sources (engine manufacturers, etc.)[1]	External
FAA/DERs	External
Senior Management	Internal
Project Management	Internal
Systems Engineers	Internal
Development Engineers	Internal
Test Engineers	Internal
Quality Engineers	Internal
Configuration Managers	Internal

Table 1: *Stakeholders*

example, all organizations want to reduce cost, shorten schedules, and lower risk. Moreover, flexibility, extensibility, and sustainability are typically considered a necessary part of every good design. Other less-tangible goals are also important. For example, customer satisfaction and market share often outweigh other cost and schedule considerations.

Next, you must quantify the identified goals. How do you quantify? To drive the return-on-investment analysis, the recommended strategy is to quantify goals in terms of cost and schedule. Cost savings are quantified by investment dollars over a specified timeframe. Schedule, of course, is quantified in calendar time – often with respect to specific milestones. Although a variety of measures can be applied to flexibility, extensibility, and sustainability, ultimately a cost and schedule impact can be identified. What is the resulting impact of alternative levels of flexibility, extensibility, and sustainability? Risks can also be addressed with cost. How much are you willing to pay to mitigate specific risks?

Figure 2: *Standard Reuse Analysis and Incorporation Process*

Although less tangible goals such as customer satisfaction and market share can also be quantified in terms of cost, much more guessing as opposed to estimating is typically involved. You may or may not want to apply costs to these factors; often, the tangible goal costs are simply compared with the understood need for these elements.

While other measures can also be effective, performing the additional analysis and estimation to assign costs facilitates the prioritization and ultimately the selection of competing strategies based on return on investment.

Prioritize Goals

After quantifying the goals, a relative priority must be given to each. Goals that have been quantified in terms of cost lend themselves well to prioritization. Intangible goals not assigned costs can be compared with this initial prioritization. For example, the lowest-cost, barely sustainable system may eventually lead to a poor marketplace reputation that must be avoided at all costs. Comparing the cost of creating the eminently sustainable system may not be worth the investment when compared to the few defects that will ultimately be addressed – except in the customer's eyes. A *simple fix* taking an *unseemly* amount of time can destroy confidence in the software and the credibility of the manufacturer.

Update Purpose and Goals

Now that the goals have been quantified and prioritized, the purpose should be readdressed. Often, the initial goals and purpose identified should change as the tradeoffs become apparent. In fact, as the analysis continues, the goals and purpose are often adjusted again and again. Furthermore, even though the goals and purpose are assigned a baseline when the optimal approach is selected, they may change with any strategy changes or changes in business conditions.

Document Results

The final step in the goal and purpose identification is to document the analysis results:

- **Purpose:** Describe the included and excluded scope for the reuse effort.
- **Goal Priority:** Describe the priority sequence.
- **Goal Quantification:** Describe the quantification and associated rationale.
- **Long-Term and Short-Term Rationale:** Discuss the long-term versus short-term tradeoffs.

- **Assumptions:** Describe the assumptions upon which the analysis was performed, including business climate, staffing considerations, and so forth.

The relative quantifications and rationales are particularly important, so do not omit them. Often, as strategies are developed, goal prioritizations may change based on the relative ease or difficulty in implementing alternative strategies. For example, if enhanced extensibility will result in minimal cost impact but is prioritized low, revisiting the goals and prioritization may be warranted.

This documentation should be made available to all involved in developing the reuse strategy. To avoid the reuse effort taking on a life of its own (as often happens), the data should be made available throughout the reuse strategy implementation.

As indicated above, the goals and purposes may change. The update of the documentation describing the goals and priorities is crucial and is often neglected.

Catalog and Analyze Existing or Proposed Artifacts

After the purpose and goals are identified, existing or proposed artifacts must be cataloged and analyzed, and the results documented. Note that artifacts to be cataloged and analyzed may already exist or may simply be proposed. Existing artifacts are products of previous development. Sometimes, no applicable previous development is pertinent.

Cataloging Artifacts

Cataloging artifacts allows the identification of the areas of potential reuse and the relationships among them. Any data associated with previous or ongoing development are fair game for reuse. This data may include the following:

- Planning data, including the Plan for Software Aspects of Certification (PSAC), software development plan, software configuration management plan, software quality assurance plan, software verification plan, and tool qualification plan.
- Requirements data, including systems and high-level software requirements.
- Design data, including the software architecture and low-level software requirements.
- Verification data, including test cases and procedures, analysis, review records, tool qualification data, and problem reports.
- Configuration data, including software configuration index, the software life-cycle configuration index,

and the software accomplishment summary.

The software configuration indexes and the software life-cycle environment configuration indexes for the potential reuse sources are excellent places to start gathering information. These documents serve as a *central clearinghouse* of information and describe directly and indirectly most of the configurable data associated with the potential source. Be aware, however, that not all reusable data are included in the configuration indexes: Configuration management and quality assurance records may not be identified in these indexes and may still be targeted for reuse.

The cataloging effort entails documenting the possible reusable artifacts and the relationships between them. Depending upon the complexity, a spreadsheet or a small database may be appropriate. You should include the following:

- **Artifact Identification:** Identify the artifacts at a useful level of abstraction. Describing each individual code file in a configuration may not be useful; groupings of files may be more appropriate (for example, low-level discrete input/output [I/O]). Although the level of abstraction should be related to the potential reuseability, ultimately the individual configurable items must be identifiable.
- **Artifact Version:** Identify the appropriate version of each artifact. When artifacts are grouped, a version identifier appropriate to the grouping should be used; for example, low-level discrete I/O associated with Release 2.2. Note that different versions of the same artifact may be applicable to the reuse analysis. This is because, over time, different versions with different desirable functionality may have been produced.
- **Related/Traced Artifacts:** The related/traced artifacts are central to certification reuse. In certifiable configurations, most artifacts are specifically related/traced by version. The ability to reuse artifacts both horizontally and vertically increases reuse benefit. Relations are described by traceability documentation, association in configuration documentation (configuration indexes), and/or applicability.
- **Configuration Location:** The physical location of the configured elements must be documented. If a configuration management system such as Clearcase or SourceSafe is used, the

location within the hierarchical configuration structure should be identified. This is particularly important when artifact groupings are used; often the path within the configuration management system is used to identify the grouped artifacts.

Artifact Analysis

Now that the artifacts have been cataloged, they must be analyzed in terms of the following:

- Functional Alignment.
- Requirements Volatility.
- Previous Development Rigor.
- Maturity of Existing Artifacts.
- Targeted Platform Changes.
- Criticality Distribution.

This analysis is discussed in depth in the companion article [1]. The results of the analysis will extend the information gathered in the cataloging stage. For each artifact (or artifact grouping), the impact of these characteristics must be described.

The specific measure used in describing the characteristic levels (including percentage, high/medium/low, relevant/not relevant, and so forth) will depend upon the needs for gradation. These measures must be selected before the analysis begins and must be consistently applied. For example, for the I/O low-level drivers, the functional alignment might be 100 percent, the requirements volatility low, the previous development rigor high (as would be expected in a previously certified product), the maturity of the existing artifacts high, no targeted platform change effects, and no partitioning for criticality distribution appropriate.

Compare and Contrast Alternative Approaches

After the artifacts are analyzed, the target software architecture must be addressed. The goal of the architecture analysis is to develop and document a number of what-if architectural candidates. To achieve this goal, the existing software architecture must be analyzed to determine how easy or difficult the incorporation into a new application will be. Two formal software evaluation processes are considered below.

Software Architecture Analysis Method

The Software Architecture Analysis Method (SAAM) [2] is simple, easy to learn, and does not require a great deal of training. The stakeholders generate a number of scenarios that describe possible future system modifications that can address the purpose and goals identified.

Description	Direct/Indirect[2]	Required Changes	Number of Changed/Added Components	Effort for Changes (estimate)
Establish lower-criticality partition for maintenance monitoring functionality.	Indirect.	Additional system safety assessment effort. Updates to architecture to enforce partitioning. Updates to PSAC. Restructuring test documentation.	Plans, requirements, design description, implementation (15 modules) test cases and procedures (25 test cases, 57 procedures) updated. Formal final testing included with overall release.	4 person months.

Table 2: *SAAM Scenario Evaluation*

Scenarios are short statements describing the interaction of one of the stakeholders with the system. Partitioning maintenance monitoring from flight-critical functionality is an example of a SAAM change. The associated SAAM scenario evaluation is illustrated in Table 2.

A number of inputs and outputs are required to perform the SAAM. One or more documented architectures are used and modified to describe potential scenarios. These documented scenarios provide a context within which the accommodation of the purpose and goals are evaluated. The primary SAAM outputs are the adjusted scenario architectures and the estimates of the anticipated costs and associated schedule. In addition, the analysis provides a greater understanding of the system functionality. Finally, the SAAM-based evaluation provides social benefits, as stakeholders come together and gain a common understanding of the costs and benefits of competing approaches.

Architecture Tradeoff Analysis Method

The Architecture Tradeoff Analysis

Method (ATAM) [2] reveals both how well an architecture satisfies particular quality goals and also provides insight into how those quality goals interact with each other.
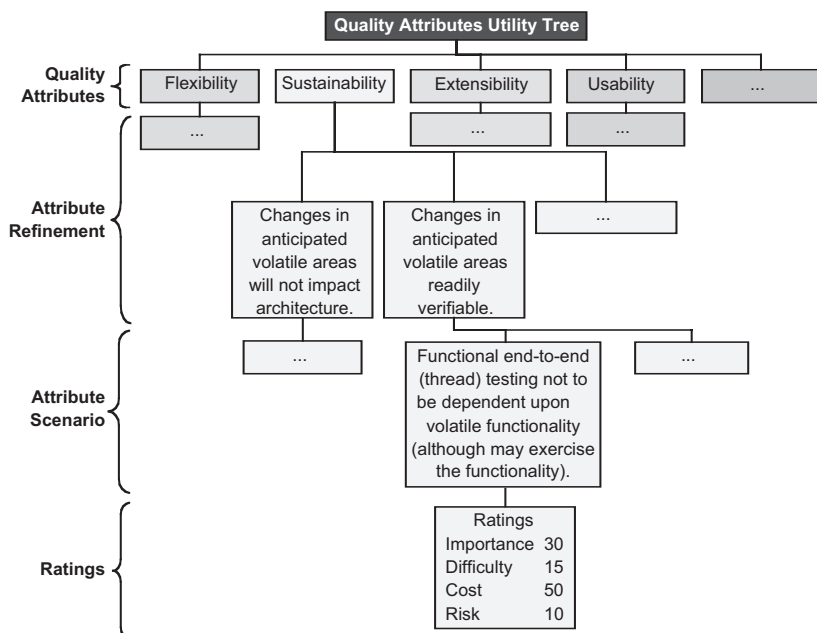
In contrast to the SAAM method, which focuses primarily on modifiability, the ATAM method provides greater insight into the quality goals: the *ilities* (flexibility, extensibility, sustainability, and so forth). The quality attribute utility tree allows a prioritization of quality attributes realized as scenarios. This tree focuses the analysis on the scenarios that address the quality attributes and ensures that the scenarios that are important to the stakeholders are addressed.

A quality attribute utility tree is illustrated in Figure 3. Typically, however, the quality attribute tree is documented with a matrix rather than with the tree format shown.

Also vital to the ATAM method are the types of scenarios addressed in the following:

- Use-case scenarios that address typical current stakeholder usage scenarios.
- Growth scenarios that address antici-

Figure 3: *Quality Attributes Utility Tree*

pated changes to the system.

- Exploratory scenarios that address extreme changes expected to stress the system.

Addressing these scenarios allows changes to the system to be considered up to and beyond the current boundary conditions of the current design.

Selecting the Appropriate Architectural Method

So which approach should you choose? Clearly, SAAM is a smaller, simpler, less expensive approach, whereas, ATAM is larger, more complex, and more expensive. Three factors should drive the selection: the complexity of the existing architecture, the potential complexity of the resulting architecture, and the approximate size of the effort. If the architectures are complex and the size of the anticipated effort is large, then the cost of ATAM is well worth the additional reduction in risk. On the other hand, if the architectures are not especially complex and the effort is small (compared to a two-week effort with a dozen people), the reduction in risk may not warrant the cost.

Select Optimal Approach

At this point, the selection of the optimal reuse approach should be made, selecting from the number of *what-if* architectural candidates that have been created based on the analysis method used from the earlier section, "Compare and Contrast Alternative Approaches." Keep in mind that the entire point of reuse is to increase the return on investment (ROI) while ensuring continued business viability. You should now have documented alternative architectural approaches that can be assessed in terms of the purpose defined and the prioritized goals. As with the initial identification of goals and purpose, all stakeholders should participate in the approach selection process. Moreover, the goal of approach selection is agreement and buy-in – all the more reason to ensure that all stakeholders are involved.

The selection of the optimal approach and the associated ROI analysis considers the following:

- Determining cost incurred over time is perhaps the most difficult part of the ROI analysis, especially accurately determining the long-term versus short-term benefits – and then making the appropriate tradeoffs. As a rule of thumb, the more work performed *up front* in emphasizing maintenance, extensibility, and flexibility, the less follow-on work will be required. The

best way to model this comparison is to plot anticipated expenditures over time and compare those expenditures to the company's financial capabilities and the overall business schedule considerations.

- Schedule considerations are typically driven by either the schedule imposed by customers, or by market conditions and the need to field products before your competitors. Although schedules imposed by customers tend to be immutable, more schedule flexibility is typically available with marketing strategy tradeoffs. In both cases, the earlier the reuse strategy is considered and implemented, the more schedule flexibility will be available in both cases. Clearly, development of a new reuse library with elements designed

> *"Determining cost incurred over time is perhaps the most difficult part of the ROI analysis, especially accurately determining the long-term versus short-term benefits – and then making the appropriate tradeoffs."*

for reuse is inappropriate for a slight change to an existing certified configuration on a tight schedule. On the other hand, if the change is slight but the schedule is not critical *and* there are other uses for a reusable library (for example, future families of similar products that would benefit from the effort), the migration of a known application may be the ideal means to introduce the reuse.

- If the reuse strategy implementation is to be funded internally by company investment, the work can be performed to align to projected needs. Otherwise, the work must be performed within the schedule permitted by the customer. Note that the risk both increases and decreases with investment-funded reuse implementation. Clearly, schedule risk decreases; however, the risk of losing funding

partway through the reuse implementation and resorting to *fast-and-dirty* development increases.

- The costs associated with reuse must be carefully estimated and documented – especially with design for reuse. Too often, manufacturers either underestimate or shift funding away from design-for-reuse activities. As a result, many of these efforts fail. The software that was intended for reuse turns out to be only applicable for a single use, but that software was more expensive because of the aborted *reusability* work. In fact, scavenge reuse is often the product of aborted design-for-reuse efforts. Unfortunately, when the reuse library does not materialize and the costs associated with each scavenge reuse instance exceeds the initial estimates based on design for reuse, reuse is given a bad name.

- Effects on intangible factors must be considered. Is the best always the lowest cost? Intangible benefits often cause more expensive approaches to be selected over less expensive approaches. Company limitations on available funding often stand in the way of the most efficient approaches as well. Therefore, the short-versus-long-term analysis is crucial to the approach selection. You must also consider the lifespan and anticipated breadth of applicability of the reuse elements. The lifespan of an application concerns the longevity, with changes, for a particular configuration. The breadth of applicability concerns the number of different applications that will make up the reuse target family.

- Plausibility and associated risk are a concern. Selecting the optimal approach is to judge the various approaches in terms of the identified purpose and the prioritized goals. When it is not possible to select among the candidate scenarios and still attain the purpose and goals, the goals and purpose must be adjusted or new scenarios generated. Risk also must be considered when selecting among competing architectures and may require the adjustment of the goals and purpose. An informed decision must be made on how much risk the organization is willing to carry. You must ensure that reasonable expectations are set.

Plan, Plan, Plan
Reuse planning is key to the success of

reuse. Effective reuse planning always includes hardware considerations; therefore, a considerable amount of planning and analysis will occur *outside the realm of traditional software plans*.

Two types of reuse planning are necessary: certification planning and project planning. Certification planning encompasses creating the planning data required by the certification authorities (typically to meet RTCA DO-178B data guidelines). The project planning concerns the internal schedules, budgets, and staffing considerations.

The key certification-planning document for reducing reuse risk is the PSAC. The strategy by which reuse is to occur, including especially partitioning considerations, should be provided to the FAA in the PSAC as early as possible to prevent costly project missteps. Another critical document for reducing reuse risk is the configuration management plan. The configuration of reusable components and the tracking of changes among different reuse instantiations are often neglected and can impact both cost and schedule (as well as embarrassment).

An additional certification-planning consideration concerns the FAA/DER used. When incorporating reuse, try to use the same DER for all reuse instantiations to allow him/her to become comfortable with the reuse data and process.

Because much of the analysis data will flow directly into the detailed project planning, the data must be realistic. Tracking against unrealistic expectations ensures failure. Many successful reuse efforts can be viewed as failures because the documented expectations were unrealistically high. Other reuse efforts fail because they are abandoned when early tracking data deviates from unrealistic plans.

Strategy Modification

As indicated above, reuse strategies can be changed mid-stream. Changes can occur based on changes in business conditions (especially funding), progress not matching plan, and so forth. When strategies change, the documented reuse analysis data is key in determining the next best or appropriate alternative path. All stakeholders should be involved in the decision to change strategies to first determine if strategy should change and then, if necessary, to determine how the strategy should change. The same consideration applied to initial strategy selection should be applied to strategy changes.

When strategies are modified, the revised purpose and goal expectations must be documented and communicated. Reuse must be tracked against the appropriate plan; otherwise, even though reuse provided meaningful savings (albeit less than initially expected), the reuse process will lose credibility.

Risk Mitigation

Reuse risk mitigation includes reigning in the scope of the reuse activity, overcoming the *not-invented-here* mindset, and avoiding (where practicable) *bleeding-edge* technology. Furthermore, a clear purpose and goals, a solid analysis, careful planning, and stakeholder buy-in mitigate the risk of reuse.

These pitfalls represent the common major challenges that you will face when implementing reuse. A myriad of other programmatic and technical risks specific to the particulars of the product and companies will plague the reuse effort. These risks include both normal development risks plus risks associated with the reusable aspect of the development.

The target of the risk process may be different, spanning many products instead of a single product. Schedules and milestones must be coordinated: Risk of delay in one product instantiation affecting the schedule and milestones of another product instantiation must be addressed. Moreover, you must address the misalignment risks associated with planning and executing the reuse-specific tasks that could include separate plans, requirement documents, design documents, and so forth.

Conclusion

Cost and schedule time can be saved and safety can be enhanced with reuse for DO-178B certifiable software. To attain maximum reuse benefits, however, you must be rigorous in your approach to the planning, analyzing, execution, and tracking of reuse. This article outlines a rigorous reuse process that provides a road map to reuse success. Keys to reuse success include the following:
- Involving the appropriate stakeholders throughout the reuse analysis and incorporation process.
- Identifying clearly the reuse goals and purpose.
- Performing a rigorous architectural analysis.
- Planning and tracking in detail reuse execution.
- Reviewing and documenting any necessary mid-stream strategy changes.
- Applying risk mitigation to the reusable aspects of development.

Addressing these key issues will allow you to take control of the success or failure of your reuse effort and, ultimately, to control your company's bottom line and continued competitiveness.◆

References

1. Lougee, Hoyt. "Reuse and DO-178B Certified Software: Beginning With the Basics." CROSSTALK Dec. 2004 <www.stsc.hill.af.mil/crosstalk>.
2. Clements, Paul, Rick Kaman, and Mark Klein. Evaluating Software Architectures. 1st ed. Addison-Wesley, 15 Jan. 2002.

Notes

1. In this example of stakeholder breakdown, major aircraft sub-system suppliers (for example, engine manufacturers) are the first-tier suppliers to the airframers. The software organization discussed would be part of a second-tier supplier, supplying components directly to the first-tier suppliers. Often, software supply sources populate even lower tiers in the supply chain.
2. Direct scenarios are currently satisfied by the system architecture; indirect scenarios require a modification of the architecture.

About the Author

Hoyt Lougee is the engineering manager, Aerospace Division, at Foliage Software Systems. Foliage delivers DO-178B process and technology consulting, custom software development, and independent verification and validation. Lougee's responsibilities include program management and software process improvement. Previously with AlliedSignal/Honeywell, Lougee has more than 13 years of experience with both military (DOD-STD-2167A) and commercial (RTCA DO-178B) aviation software development and certification efforts. Lougee has authored a number of white papers and presented at the 2002 Digital Avionics Systems Conference.

Foliage Software Systems
168 Middlesex TPKE
Burlington, MA 01803
Phone: (781) 993-5500
Fax: (781) 993-5501
E-mail: hlougee@foliage.com

Opening Up Open Source

Michelle Levesque and Jason Montojo
University of Toronto

As Free/Libre/Open Source Software (FLOSS) becomes an increasingly popular alternative to commercial software, its user base has extended from software developers to the general public. However, many FLOSS projects still suffer from usability issues such as nonintuitive interfaces and poor documentation. Many of these problems stem from the technical elite's general impatience towards new users. This negative attitude causes less effort to be spent on making it easier for new users to use the software. Thus, many of the non-technical aspects of the development cycle such as documentation and interface design are neglected. This neglect is further emphasized by the fact that many developers would rather work on the code than documentation and interface design. These social and usability issues must be resolved for FLOSS to become a much more viable alternative for both technical and non-technical users alike.

Free/Libre/Open Source Software (FLOSS) has become a competitive alternative to commercial software in almost all areas of computing. It has become more and more common to find FLOSS software in schools, hospitals, governments, businesses, and homes. The Linux operating system, the Firefox Web browser, and the Apache Web server are just some examples of FLOSS software that is being used on an increasing basis. There are many obvious advantages to FLOSS: the freedom to tinker with the code, the ability to observe exactly what the software is doing, and the lack of dependence on a commercial provider. As well, FLOSS has a strong international community of programmers who volunteer countless hours to produce this software.

With so many obvious advantages, the problems with FLOSS are often overlooked. Though the FLOSS movement claims to be open to everyone, many users and developers alike feel that the community is not as welcoming as it claims to be. This exclusion creates a gap between users and FLOSS developers, and this gap fosters software usability problems.

It is becoming increasingly common for corporations to participate in FLOSS projects; however, in this article we are focusing on issues that arise from projects produced primarily by self-governed volunteers. This is both because corporate software development is already well researched, and because it is the strong volunteer community that makes FLOSS so unique.

Open Community

One of FLOSS' greatest strengths is its openness: Anyone is free to contribute. However, more than just the capacity for contribution is necessary to create an open community. FLOSS communities are built upon geek and hacker cultures, and these cultures are not known for their friendliness towards new users or those with different opinions. This stubbornness often causes new users or non-hackers to feel unwelcome by the community. Though this exclusion is a social issue, the nature of FLOSS causes it to affect more than just social relationships. It also impacts the code.

Because FLOSS programmers tend to contribute code during their spare time, it is natural that they would make contributions that interest them. However, not all software design tasks are viewed as being equal. For example, there is much less geek prestige to be earned for interface design, user testing, or documentation. Therefore these tasks are often neglected by the volunteer coders who can receive more *geek cred*[1] by focusing on other elements of software design. An open source usability study states:

> Indeed, there may be a certain pride in the creation of a sophisticated product with a powerful, but challenging to learn interface. Mastery of such a product is difficult and so legitimates membership of an elite who can then distinguish itself from so-called *lusers*. [1]

In contrast, software companies hire employees to specifically perform tasks like user testing, documentation, and interface design. Usability experts, graphic artists, tech writers, and others all have a place in commercial software development. So where are these participants in FLOSS development? The hard geek culture behind FLOSS may be useful in creating powerful software, but it also drives away these other essential members of the software team [1].

When users criticize FLOSS' usability, their suggestions are often ignored or flamed, rather than analyzed and used to improve the software: "Geeks tend to treat others who disagree with loud, obvious disdain. These behaviors are harmful both to the disagreer and to the community as a whole" [2]. Many users are told that they have no right to complain about FLOSS software unless they are willing to fix it themselves [3]. This attitude chases off many users who might otherwise have become firm FLOSS supporters. In turn, it reduces the number of users who are available to report bugs, participate in user testing, and help out with basic documentation. Thus the gap between user and developer widens.

The distributed nature of FLOSS also means that not all developers have the same goals in mind. Some developers participate in order to create superior software. Others simply want to tinker with some code. Increasing the software's accessibility is not a priority for the second group. They are creating software for their own use, or simply coding for coding's sake, rather than creating software for a general audience. As one programmer on Slashdot <www.slashdot.org> said:

> You'd better believe I'm designing it for ME. It's not fun to design programs for other people. That's a job. I wouldn't do that for free. If you would like to PAY me to make it work for you,

I would be happy to. [4]

This attitude draws effort away from software usability and often causes users to believe that FLOSS is intended for hackers alone.

Usability

The engineer's fallacy is the belief that if something is easy for the designer, it will be easy for the average user, too. Usability does not occur naturally in software, it is something that must be consciously planned. But most software written outside the industry tends to be written by programmers for their own use, so they tend to focus on the technical aspect of their program such as the language it is written in or the algorithms being used. Some of them try to show off their mastery of the technology to other hackers, oftentimes at the expense of usability [1]. Although they "can be very good at designing interfaces for other hackers, they tend to be poor at modeling the thought processes of the other 95 percent of the population" [5].

When interfaces fail at being intuitive, users have two main options for getting help: developer-provided documentation, and community-provided documentation such as forums and mailing lists. The problem with developer-provided documentation is that it is sometimes non-existent, especially in FLOSS projects. When it is there, it is typically written with the assumption that the reader already has some technical background about the software and what it does. Information on forums and mailing lists is often just as technical as developer-provided documentation since the main code contributors are usually the ones fielding the questions.

FLOSS' community-provided documentation also assumes a certain level of technical skill. Unlike commercial software, which usually provides a hotline to satisfy this need, FLOSS projects typically use online communication channels like e-mail, newsgroups, chat rooms, online forums, and mailing lists [6]. Although these are invaluable resources, they are not easily accessible to the average user [7] who may not know how to use them or even that they exist.

Providing adequate documentation and easily accessible assistance is not an easy task, and many FLOSS developers would rather focus on their work than deal with these issues: "We all know

that some projects, probably most, need better documentation and could use some more refactoring. But until you open up your wallet, get off our backs" [8]. In an environment where only hackers feel comfortable, yet do not want to do certain tasks, these tasks quickly become neglected.

To make matters worse, the distributed nature of FLOSS means that the different contributors may have different ideas about where they want their project to go. Such conflicts end up producing "15 different editors, several different Web browsers, several different desktops, and so on" often leaving the end-user with more choice, and just as much confusion [3]. For every choice that the user must make, the FLOSS

"Though the FLOSS movement claims to be open to everyone, many users and developers alike feel that the community is not as welcoming as it claims to be."

---

learning curve becomes that much more difficult. Though there are many advantages to the variety of forks available in FLOSS, it just adds another layer of complexity for the users.

Conclusion

None of the problems that we describe above are that impossible to resolve.

> The existence of a problem does not necessarily mean that all OSS [open source software] interfaces are bad or that OSS is doomed to have hard-to-use interfaces, just a recognition that the interfaces ought to be and can be made better. [1]

Usability is an issue that has not been solved in proprietary software either. However, the FLOSS community has to actively acknowledge that this problem exists before an effective resolution can be implemented.

There are already some initiatives in place to try to solve some of these

problems. An example is GrokDoc <www.grokdoc.net>, a usability study that strives to create documentation for GNU/Linux. Unlike most documentation, which is created by having developers explain how to perform various tasks, GrokDoc is based on having new users demonstrate exactly what they find difficult.

Efforts are also being developed to deal with the unwelcoming environment that many users and developers feel exists in FLOSS communities. The most pronounced of these efforts are the support mechanisms being built to try to encourage women to participate in FLOSS activities. FLOSSpols <www.flosspols.org> is a study currently in progress to try to understand the wide gender gap in FLOSS and to offer concrete recommendations on how to solve this gap. Other efforts include WOWEM, a gender equity and FLOSS research and education project, and LinuxChix, a community for supporting women in Linux. Despite these efforts, there is still a strong belief that most geek-saturated communities like Slashdot are often unwelcoming and hostile environments.

It is important to remember that volunteers do most FLOSS programming. It would be unreasonable to ask these volunteers to contribute in ways that they find boring, tedious, or work-like. However this does not mean that there will always be tasks that are neglected in FLOSS development.

We believe that if the FLOSS community makes the social adjustments necessary to create a more open setting, then non-hackers will become more inclined to participate. This includes graphic designers, teachers, writers, more developers, and just normal users. It is the inclusion of all of these groups in the development process that will make FLOSS stronger, more usable, and truly open to all.◆

References

1. Nichols, David, and Michael Twidale. "The Usability of Open Source Software." First Monday 8.1 (2003) <www.firstmonday.dk/issues/issue8_1/nichols>.
2. Lester, Andy. "Geek Culture Considered Harmful to Perl." Lightening Talks at Yet Another Perl Conference, St. Louis, MO, 20 June 2002 <www.petdance.com/perl/geek-culture>.
3. Gunton, Neil. "Open Source Myths." 25 July 2004 <www.neil

gunton.com/open_source_myths>.
4. <u>Slashdot</u>. "Five Fundamental Problems With Open Source." Comment By Bill Shooter of Bul. 13 Apr. 2004 <http://ask.slashdot.org/askslashdot/04/04/12/1757244.shtml>.
5. Dibona, C., et al. <u>Open Sources: Voices From the Open Source Revolution</u>. 1st ed. O'Reilly, Jan. 1999 <www.oreilly.com/catalog/opensources/book/raymond2.html>.
6. Cubranic, Davor. "Open-Source Software Development." University of British Columbia, Nov. 1999 <http://sern.ucalgary.ca/~maurer/ICSE99WS/Submissions/Cubranic/Cubranic.html>.
7. Trudelle, Peter. "Bugzilla Bug 89907 - Need to make it easier for users to make us their default browser." Mozilla, 2 Jan. 2002 <http://bugzilla.mozilla.org/show_bug.cgi?id=89907#c14>.
8. Countryman, Dan. "Sometimes great things have too [sic] be absorbed and thrown way [sic]." Object Country, 25 Apr. 2004 <http://jroller.com/page/objectcountry/20040425>.

Note

1. Credibility among young fashionable urban individuals.

About the Authors

Michelle Levesque is currently involved with the Citizen Lab where she designs and implements programs to enumerate and circumvent state-imposed Internet content filtering. She is working towards a degree in software engineering at the University of Toronto.

E-mail: ml@cs.toronto.edu

Jason Montojo is currently developing bioinformatics Web application software for the Blueprint Initiative in Toronto, Canada. He also has worked on the Eclipse Open Source project as a member of the platform development team at Object Technology International. Montojo is working towards a degree in software engineering at the University of Toronto.

E-mail: j.montojo@utoronto.ca

LETTER TO THE EDITOR

Dear CROSSTALK Editor,

It will become increasingly clear to policy-makers over the next several years that the open source paradigm must be embraced by the Department of Defense (DoD). Despite the compelling arguments for open source and broad commercial industry acceptance, there will be significant government structural and defense industry interest impeding the inevitable. These impediments will likely only fall when one or more major defense acquisition programs (MDAPs) fail due to software cost overruns and schedule delays, affecting the bottom line of major defense contractors. However, determined policy efforts could accelerate the adoption of open source, saving the DoD billions of dollars, reducing delays, and facilitating joint operations through interoperability.

The size of the effective source lines of code (SLOC) for MDAPs has reached well over 10 million SLOC and is growing rapidly. The development cost of this software, which the Office of the Secretary of Defense (OSD) Cost Analysis Improvement Group (CAIG) puts at more than $300/SLOC, is only decreasing slowly as increasing complexity competes with process and tool improvements. As a result, software development is quickly approaching 50 percent of the cost of MDAP development, and the eight years or more of development schedule has placed software permanently on the critical path to deployment. Software has become the cost- and schedule-limiting factor of our future defense capabilities.

Many of the requirements driving software have and are being addressed by many programs, yet data suggests only a (roughly) 30 percent savings is achieved by reusing and modifying existing code. Code reused from other programs must often be substantially modified to adapt to the architecture and specific requirements of another program. Architectural differences between programs impede coarse-grained reuse while differences in the details of requirements require modification of code at the fine grain. Reuse is further hampered by the concurrent evolution of the programs being reused. Programs committed to reuse as a cost saving measure can establish program-to-program memorandums of agreement, but it is easy to see how this would become an n-factorial interface coordination problem when looking across multiple programs, with significant duplication of effort.

A current alternative approach to open source is for software reuse to be mandated by policy and the establishment of joint development efforts such as the Mission Planning System Framework and Common Capabilities being developed by the Air Force and Navy. Unfortunately, *not-invented-here* concerns have prevented widespread assumption of this important command-and-control software initiative. As far as I know, no MDAPs have actively participated in the requirements definition of the Common Capabilities or adapted their command and control architecture to leverage or influence this development.

The open source development paradigm offers the promise and presents the challenges of coordinated definition of requirements, architecture and design approach, coding, and testing. The promise is enhanced reuse and interoperability, shared development costs, and shortened development schedules. The challenges, among others, are security, organizational and contractual performance responsibility, and proprietary and licensing considerations. These challenges should be studied and can be overcome. Many of them have been and are being addressed in a variety of ways by the commercial open source community, which now includes the largest commercial software vendors, including IBM, Novell, Sun Microsystems, and even Microsoft.

The development and advocacy of modern and flexible open architectures and standards by the Defense Information Systems Agency is a necessary precondition for guiding the development of potentially hundreds of open source projects within the DoD so that the software for multiple MDAPs can be more easily assembled from open source code. Once security and other key issues have been addressed, I recommend the establishment of (including ground rules for) a repository for DoD open source software development, modeled in some ways along the lines of the popular <www.sourceforge.net> open source development Web site. Perhaps the URL might be <www.sourceforge.mil>. The force of arguments for open source are so compelling that early adopters in the DoD will find many reasons to utilize such a repository once it becomes available. Among other development artifacts, requirements databases, unified modeling language models, source code, and bug resolution tracking should be high priorities for such a repository.

In the meantime, I encourage the DoD software acquisition community to become more aware of the open source development. Junior officers and developers might want to select one of the more than 50,000 projects at <www.sourceforge.net> to participate in. Programming skills are not necessarily required. One can contribute in many ways, including software testing and documentation. Software acquisition managers could consider how to leverage existing open source software in the programs they manage. Policy-makers could engage industry leaders that have recognized the marketplace advantages of open source to address unique DoD concerns. I hope that open source advocates will spread across the software acquisition community and important DoD software institutions that include Carnegie Mellon University's Software Engineering Institute, the U.S. Air Force's Software Technology Support Center, and the MITRE Corporation.

Finally, I hope that the defense industry will recognize that the open source paradigm can be tailored and leveraged into increased competitiveness, productivity, and profits, while delivering more capability to our military faster and cheaper.

Thomas M. Schaefer
Senior Defense Cost Analyst and Software Developer

2005 CROSSTALK EDITORIAL BOARD

CROSSTALK proudly presents its 2005 CrossTalk Editorial Board. Two technical reviewers, in addition to both the publisher and associate publisher, review each article submitted to CROSSTALK . Most reviewers on the list below have graciously volunteered their own time to support CROSSTALK's technical review process. We give a very special thanks to all those participating on our 2005 CrossTalk Editorial Board.

– Tracy L. Stauder, Publisher

High Stakes and Misdemeanors

So now you are a technical program manager. You climbed from trainee, senior engineer, team lead, project manager to program manager. You planned your career and made the sacrifices.

When staff set up the office pool, you went back to night school. While colleagues went to the local bar, you were in class with the statistics tsar.

While teammates looked for the easy chore, you volunteered for the quality program de jour. When colleagues were on the slopes, you kept the project off the ropes.

While your rival hatched political schemes, you were leading tiger teams. While the boss was on the links, you discovered how the customer thinks.

You endured staff meetings, suffered cubical seating, and dodged performance review beatings.

As you don your coveted title of the Geek Godfather, you will realize the job is poles apart from your aspiration. Even the best-prepared engineer can be blindsided by the realities and limitations of the job. The stakes are high and rife with risk. Here are a few misdemeanors to avoid.

First, you have little time to run the program. Even though you are the Big Kahuna, the daily work is now out of your hands. Your time and influence will shift from direct to indirect: articulating and conveying strategy, institutionalizing rigorous processes, and setting value and tone for projects – not the typical skills of an engineer.

Ironically, the transition from engineer to program manager leaves a sense of lost control. Initially, you feel more like Seinfeld's Kramer – restless, disjointed, and sketchy – than like his alter ego Peter Von Nostrand: cool, calm, and collected. New program managers tend to gravitate back to the comfort and familiarity of daily operations at the expense of mounting strategic, financial, legal, personnel, and stakeholder demands.

It is critical that you learn to relinquish responsibility and manage through delegation and accountability. Like the bridle, the keel, and the fulcrum – it's about leverage. Without leverage, you will lose control.

Second, you are always sending signals. The high profile of a program manager is viewed as a perk of the job. Au contraire. The extent of scrutiny and interpretation of your every move can be vitiating. The stealthy days in the lab, computer room, or office are gone. Your microphone is always on and the cameras constantly rolling.

Also gone are speculative discussions with managers, employees, and the public. One day you explore the intrigue of open source software and the next day you wake up with a new Linux server farm. One day you complement the use of rate monotonic analysis and the next day you are listening to a briefing on vacation scheduling via rate monotonic analysis.

Consider carefully your actions, conversations, and messages. Strive for simplicity, clarity, consistency and master analogies, metaphors, and allegories to communicate your message.

No Soup for You!

Third, beware of shooting stars. Like the grass on the other side, it is tempting to reach for another's guru. Do not be blinded by that light. Shining stars in one environment fade in others. Ask the Yankees about Alex Rodriguez's playoff performance. Moreover, stars do not stay with organizations long. Supernovas that jump, like free radicals, to your program are susceptible to other enticements. Ask the Cleveland Cavaliers where Carlos Boozer is playing this year.

Bringing in a superstar resembles an organ transplant. The new body rejects the prized organ. This battle consumes resources that take away from the healthy parts of the body that soon cause other health problems. Transplanting a star into your organization will no doubt cause resentment, conflict, and impede team morale.

My advice: grow your stars from within. Internal stars know the culture, garner employee support, and are more loyal. If you do star search, assure the luminary can shine in your program.

Finally, issuing commands can be costly. The consequences of orders expand proportionally to the breadth of command. Unilateral commands that overrule thoughtful decisions trigger resentment, insecurity, and perplexity. Excessive intervention, inquisition, and supersession create bottlenecks as employees are excessively inclined to consult you before acting.

As program manager, you will have to make decisions and give orders. When doing so, be selective, deliberate, and inclusive with a broader plan of action in mind. If not, your office will resemble the lines at Seinfeld's famous Soup Kitchen – no funding for you! Next!

–Gary Petersen
Shim Enterprise, Inc.

CrossTalk / MASE
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820